HAL
open science

Certification of static analysis in many-sorted first-order logic

Pierre-Emmanuel Cornilleau

To cite this version:

Pierre-Emmanuel Cornilleau. Certification of static analysis in many-sorted first-order logic. Other [cs.OH]. École normale supérieure de Cachan - ENS Cachan, 2013. English. NNT : 2013DENS0012 . tel-00846347

HAL Id: tel-00846347

https://theses.hal.science/tel-00846347v1

Submitted on 19 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

THÈSE / ENS Cachan - Bretagne
sous le sceau de l'Université Européenne de Bretagne

pour obtenir le grade de
DOCTEUR DE L'ÉCOLE NORMALE SUPERIEURE DE CACHAN
Spécialité : Informatique
École doctorale : MATISSE

présentée par
Pierre-Emmanuel Cornilleau
Préparée au centre Rennes - Bretagne Atlantique de l'Institut National de Recherche en Informatique et en Automatique

Certification of Static Analysis in Many-Sorted First-Order Logic

Soutenance prévue le 12 mars 2013
devant le jury composé de :

Andy M. King
Reader, University of Kent / *Rapporteur*

Stephan Merz
Directeur de recherche, INRIA Nancy / *Rapporteur*

Viktor Kuncak
Associate professor, EPFL / *Examinateur*

Olivier Ridoux
Professeur, Université de Rennes 1 / *Examinateur*

Thomas Jensen
Directeur de recherche, INRIA Rennes / *Directeur*

Frédéric Besson
Chargé de recherche, INRIA Rennes / *Encadrant*

Contents

Contents **i**

1 Introduction **1**
1.1 Program analysis and verification 2
1.2 Who watches the watchmen 3
1.3 Static analysis result certification 4
1.4 Our thesis 6
1.5 Outline of the dissertation 7

2 Abstract interpretation **10**
2.1 Abstract interpretation's core mechanisms 10
2.1.1 A theory of approximation 11
2.1.2 Building new domains 12
2.1.3 Formalising a static analyser 12
2.2 Core numerical language & semantics 13
2.2.1 Syntax 13
2.2.2 Semantics 15
2.3 Numerical analyses 19
2.3.1 Interval analyses 19
2.3.2 Polyhedron analyses 20
2.3.3 Octagon analysis 22
2.3.4 Conclusion 24
2.4 Core object-oriented language & semantics 24
2.4.1 Syntax 24
2.4.2 Semantics 26
2.5 Object-oriented analyses 30
2.5.1 Simple analyser : ByteCode Verifier 30
2.5.2 Null pointer analysis 32
2.5.3 Conclusion 34

3 Deductive verification background **35**
3.1 Deductive verification 35
3.1.1 Axiomatic semantics 35

3.1.2 Weakest precondition calculus 38
3.2 Deductive verification of modern languages 39
3.2.1 Memory model 39
3.2.2 Framing 40
3.2.3 Intermediate Verification Languages 41
3.2.4 Automated deductive verification 42
3.3 Establishing trust 44
3.3.1 Trusted Computing Base 44
3.3.2 Proof-Carrying Code 45
3.3.3 Foundational Proof-Carrying Code 47

4 S.A. result certification 49
4.1 Approach 49
4.1.1 Overview of the approach 50
4.1.2 Applications 52
4.2 Specification language 53
4.2.1 Many-sorted FOL 53
4.2.2 Theory of semantic states 55
4.3 Methodology for the result certification of a static analyser . 56
4.3.1 Compiling abstract states into assertions 57
4.3.2 VCgen methodology 58
4.3.3 Handling semantic invariants 60
4.4 VCgen for core languages 61
4.4.1 Numerical VCgen 61
4.4.2 Object-oriented VCgen 66
4.5 Conclusion 71

5 V.C. soundness & numerical S.A. certif. 72
5.1 The Why3 intermediate verification language 73
5.2 Relying on an I.V.L. for V.C. gen. and V.C. soundness 74
5.2.1 Overview of the implementation of the approach . . . 75
5.2.2 VC generation 76
5.2.3 Small step semantics as an interpreter in the IVL . . . 78
5.3 1st experiment: result certification of numerical analyses . . . 81
5.3.1 Numerical language VCgen 82
5.3.2 Experiment 82
5.3.3 Results 83
5.4 Conclusion and related work 85
5.4.1 Trusted Computing Base 86
5.4.2 Certified static analysis 87
5.4.3 Limits 88

6 Simplified V.C. for analyses of the heap 89
6.1 Restriction to a family of analyses 89
6.1.1 Instrumented semantics 90
6.1.2 Parametrised analyses 92
6.1.3 Instantiations of the framework 93
6.2 Simplification of object-oriented VCs 95
6.2.1 A fragment with quantifier elimination 95
6.2.2 Decidable verification conditions 96
6.3 2nd experiment: result certification of object-oriented analyses 103
6.3.1 Object-oriented language VCgen 104
6.3.2 Generation of verification condition revisited 104
6.3.3 Results 105
6.4 Conclusion 106
6.4.1 Further work 107

7 S.M.T. background 109
7.1 Lazy SMT approach 109
7.1.1 SAT modulo theory 110
7.1.2 Multi-theory D.P. for conjunctions 113
7.1.3 Optimisation of lazy SMT 114
7.2 Checking SMT proofs 114
7.2.1 Proof producing D.P. 115
7.2.2 Proof verification 115

8 S.M.T. result certification 118
8.1 S.M.T. result certification in Coq 118
8.1.1 Result certification in Coq 118
8.1.2 Overview of the approach 119
8.2 Proof system 121
8.2.1 S.M.T. proof format 121
8.2.2 A proof system for Nelson-Oppen with case-split . . . 123
8.3 Reflexive S.M.T. proof checker in Coq 125
8.3.1 Theory interface 125
8.3.2 Nelson-Oppen proof checker 126
8.4 Experiments 127
8.4.1 Certificate generation 127
8.4.2 Sampling 129
8.4.3 Results 129
8.5 Conclusion 132
8.5.1 Comparison with other works in Coq 133
8.5.2 Further work 134

9 U.F. result certification **136**
9.1 U.F. proof producing D.P. 137
9.2 U.F. proof verifiers 138
9.2.1 Command verifier 139
9.2.2 Proof forest verifier 140
9.2.3 A verifier using trimmed forests 140
9.3 Implementation and experiments 141
9.3.1 U.F. verifiers in Coq 141
9.3.2 Benchmarks 142
9.4 Conclusion 144

10 Conclusion **145**

11 Bibliography **150**

A Certificate checking and generation for LRA and LIA **170**

List of Listings **173**

List of Figures **174**

List of Tables **176**

Chapter 1

Introduction

> The EDSAC was on the top floor of the building and the tape-punching and editing equipment one floor below. [...] It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.
>
> — Sir Maurice Vincent Wilkes, in 1985 [Wil85, p. 145]

> And finally, the cost of error in a certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war. Thus the practice of program proving is not only a theoretical pursuit, followed in the interests of academic respectability, but a serious recommendation for the reduction of the costs associated with programming error.
>
> — Sir Charles Antony Richard Hoare, in 1968 [Hoa69]

1.1 Program analysis and verification

The idea of automatic computation is quite old, and mechanical computation devices can be traced back to Ada Lovelace and other illustrious pioneers, but its present incarnation in digital computers is barely sixty years old. It quickly evolved from fully hardware machine, where programs were maps of wires and connectors, to machines combining hardware and software. Until the mid 80s, computers were programmed using punch-cards, and bugs could come from a mistake in the initial program, an undetected faulty translation from coding pad to card decks, a misplaced card, a card jammed in the reader, or a dead moth in a relay. Nowadays, it is far more convenient to execute code on a machine: programs are written on computers, as data, and can be manipulated by other programs. There are still many problems to solve to ensure no error is introduced once the program have been written. Even if no card has to be punched anymore, programs can contain millions of lines of code, be developed by many persons on a long time, be deployed on a wide variety of platforms and the odds of getting the correct computation may not seem much more favourable than in the punch-card era. The stakes have been raised too, and Hoare's statement applies more than ever, as software controls many aspects of our lives, from railway interlocking systems to air-plane flight control, banking transactions, or medical procedures.

To do proofs on programs, the first step is to give a formal definition of the language in which they are written: to define its *semantics*. Using mathematical tools and reasoning, it is then possible to *prove* theorems on the formal semantics of a program, *e.g.*, the absence of errors. Errors can be informally separated in two categories. *Functional errors* are due to differences between the specification and the code, *i.e.*, the code does not compute the correct value. *Run-time errors* on the other hand are due to the code not having a semantics, or running into error state of the semantics. The latter can be seen as a functional error too: the program does not compute, thus it does not compute the expected value. Informally, functional errors correspond to a wrong understanding of the specification, and run-time errors correspond to a wrong understanding of the semantics of the language.

The semantics of a program is a complex object itself, and detailed formal reasoning can be long and tedious, thus pen and paper proofs tend to be error prone, if feasible at all. In the same way programs are necessary to handle complex computation, programs are necessary to handle complex *reasoning* about computation. Verification environments, such as *proof assistants*, can help enforce rigorous reasoning, by being used to check each step of a proof according to the formally defined proof system, and *static analysis* provides automation or partial automation regarding tedious but simple reasoning. Rice's theorem informs us that any *interesting* property

will be undecidable for a Turing complete language, *i.e.*, if the programming language is expressive enough to write all programs that a computer can execute, it is impossible to detect automatically *exactly* the faulty programs, *i.e.*, to prove automatically the absence of errors. Even the absence of run-time errors, defined with respect to the semantics of the language and not the specification of a program, is an undecidable problem in general.

Partial proof procedures have proved very useful in practice to prove the absence of run-time errors in *some* programs. For instance, the *abstract interpretation* framework allows the formalisation of terminating modular static analyses. To enforce termination, only an *over-abstraction* of all possible executions is computed, thus the algorithm can produce *false positive*, *i.e.*, a *sound* analysis will not accept programs that result in run-time errors but it can reject programs which do not produce run-time errors. The *precision* of a static analyser depends on the precision of the abstraction domain it uses, on the definition of the operators responsible for its termination, and on its efficiency in practice, *i.e.*, the performances of its implementation on a given program. Even if the *formalisation* of the analyser can be proved sound using the abstract interpretation framework, the results of a static analyser can only be trusted as far as the implementation is supposed faithful to the formalisation.

1.2 Who watches the watchmen

In general, proofs about large programs are impossible to build and manipulate without the use of other programs, the assessment of the *trusted computing base* of a formal verification environment is a crucial issue. A common criterion is that the amount of lines of code of the programs to be trusted should be small, and the implemented algorithms should be easy to understand, such that the trusted programs can be reviewed *by hand*. But this criterion eliminates the possibility to use powerful analysers to *prove* properties, and even if a particular analyser has been reviewed and deemed trustworthy, its implementation is fixed, *i.e.*, it can not benefit from further advances—*e.g.*, new abstract domains or more efficient algorithms—without a considerable *certification* effort. To be able to use an analyser to make proofs, a solution is to prove the soundness of the whole analyser's implementation in a proof environment, if the latter is deemed trustworthy. However, this requires to develop and prove the analyser in a constrained programming environment, such as the fully functional language of a proof assistant. The constraints imposed by the proof assistant may hinder the use of optimised data structures and complicate the development of an efficient analyser. Thus the formal proof of the implementation of the analyser may require a *large* proving effort, and in the end, it does not solve the problem of updating the analyser's algorithms.

Another solution to use an analyser to make *trustworthy* proofs is to use the *result certification* methodology [Ler06]. This methodology is very general and can be applied to any decision problem. The principle is to verify the result of a decision procedure rather than its implementation, using a process deemed trustworthy. And to retain automation, rather than searching for proofs of the soundness of the results as they come, it is possible to implement a *result verifier* in the proof environment. This verifier can be made simpler than the decision procedure that comes up with the result depending on the definition of what a result is. In other words, the decision procedure is asked for more than a simple "yes"/"no" verdict: for any result, it is required to produce a *certificate* that is used to verify the verdict. For example, on the satisfiability problem of propositional formulae, an incomplete result verifier could be a model checker. If the decision procedure claims that a formula is satisfiable, it is required to produce a model as a *certificate*. The result verifier then applies the rules of propositional logic to prove that the given certificate is a model of the initial formula. Such a verifier is much simpler to prove sound than a decision procedure for propositional satisfiability: indeed, the model checking problem is linear in the size of the formula whereas the satisfiability problem is NP complete. Of course, it is an *incomplete* result verifier, because it does not define certificates for unsatisfiability results. However, using a complete proof system of unsatisfiability—*e.g.*, the resolution rule—certificates for unsatisfiability results can be defined as resolution trees, and the result verifier is in this case a proof checker. Again, the result verifier is simpler than the decision procedure and therefore easier to implement and prove in a proof environment, even if it can take a number of steps exponential in the size of the unsatisfiable formula. Applied to a static analyser, the result certification methodology requires the analyser to give more information than just "yes"/"no"/"I do not know". Instead, the output of the analyser should be enough information for the result verifier to be able to prove the verdict.

1.3 Static analysis result certification

A result certification approach has several advantages over the certification of the implementation of an analyser. The same result verifier can be used for several versions of the analyser, therefore the analyser can be used to prove properties on programs even if its implementation is not trusted, as long as the underlying logic is not changed, *i.e.*, as long as the result of the analyser can be described in the same certificate format. Furthermore, if the certificate format is general enough, the result of different analysers can be verified. In this sense, the result certification approach brings some modularity to proofs by static analysis, as it introduces a separation between proof search and proof verification: they become completely separate

process, and the proof search can be optimised without compromising the soundness of the result. The two main properties that need to be verified are i) the soundness of the results, *i.e.*, the analyser has computed a correct over-approximation and ii) its precision, *i.e.*, the computed over-approximation is precise enough to prove the absence of run-time errors. Verifying the soundness of the result ensures that even if the analyser has made some errors during the computation, they did not jeopardise the soundness of the proof. Verifying the precision of the result ensures that the final conclusion given by the analyser, with respect to the absence of errors, is sound. Another benefit of a result certification approach follows from the observation that a result verifier can be used as an oracle to test the analyser. As any program, a static analyser needs to be tested during its development. Using a result verifier as an oracle means that any program can be used as a test case for the analyser. The only limitation is that the result verifier gives no information regarding the *expected* precision of the analyser. If a program is rejected because the result of the analyser is sound but not precise enough, it may be because i) the implementation of the analyser made an error during the computation, and did not compute the best result we could expect from the formalisation, or ii) because the abstract domain is not precise enough, *i.e.*, the abstract domain is not appropriate for the property at hand, or the computations should more precise. Nevertheless, the ability to check the results of the analyser automatically when developing the analyser can be very useful, especially if the analyser needs to be optimised to produce results in a reasonable time on large programs.

Developing a result certification approach gives some leeway in choosing the format of the certificates, and that choice depends on the kind of verifier that is desired. If the certificates are empty, then the verifier is a decision procedure just as the analyser is. If the certificates are complete descriptions of proofs encoded in a proof system then the verifier is a proof-checker. But the certificates could be anything in between these two extremes, or any other kind of data. It does not have to be proof object at all: the important property is that there exists a proof that if the verifier accepts the certificate then the analyser result was sound and precise enough to prove the absence of run-time errors, but the verifier itself can perform any kind of computation as long as the previous property can be proved. For static analyser formalised in the abstract interpretation framework, an obvious candidate for certificates is the abstraction of the program computed by the analyser. However, this abstraction is not formulated in terms of usual program logic, such as Hoare logic, but in an abstract domain dedicated to the property the analyser tries to establish.

1.4 Our thesis

The result certification methodology allows the use of static analysis to prove theorems on the semantics of programs. It requires neither a proof effort as large as the formal proof of the implementation of the analyser would entail, nor compromising on the efficiency of the analyser. Many-Sorted First-Order Logic can be used as a middle ground between expressive abstract domains and generic, powerful decision procedures. It offers a framework in which the result of an analyser can be described precisely, and for which decision procedures have reached the degree of maturity necessary to make a result verifier practical. Our approach can be summarised as follows.

1. We derive from the semantics of a language and from the formalisation of an analysis a *verification condition calculus* that given a program, generates formulae in Many-Sorted First-Order Logic that are valid only if no execution of the program exhibit run-time errors. The verification condition calculus is the first part of a result verifier. The second part, needed for the verifier to be automatic, is an automated theorem prover able to discharge the verification conditions.

2. We apply the result certification methodology to Satisfiability Modulo Theory (SMT) solvers—a family of automated theorem provers well-suited to discharge Many-Sorted First-Order Logic formulae—in order to improve their reliability and justify the trusted computing base of the static analysis result verifier.

It follows that a static analysis result verifier combines a verification condition calculus and an SMT result verifier, and a certificate is composed of the abstraction of a program computed by the analysis and a set of SMT certificates for the verification conditions. In the dissertation, we detail the following contributions.

- A methodology to derive a *verification condition calculus* from an operational semantics of a language and the formalisation of a static analyser in abstract interpretation. Our experiments have shown that the verification conditions produced for numerical analyses can be discharged by off-the-shelf decision procedures.

- A verification condition calculus dedicated to a family of analyses of the heap that produces *quantifier-free verification conditions* that can be discharged efficiently by off-the-shelf decision procedures.

- A modular proof-format and *result verifier for satisfiability modulo theory solvers*, implemented in Coq, that can furthermore be used to improve the automation of the proof assistant.

- An *implementation scheme* that relies on an intermediate verification language to generate the verification conditions, to prove the soundness of the verification condition calculus with respect to the operational semantics, and to interface with automated theorem provers.

1.5 Outline of the dissertation

The analyses we consider are presented in Chapter 2. They belong to two different categories that use different abstract domains: numerical analyses and analyses of the heap. The domains of numerical analyses can be easily translated into linear constraints over numerical variables, and these constraints can be embedded in traditional program logic based on Many-Sorted First-Order Logic. Analyses of the heap, on the other hand, are based on dedicated abstract domains. If the certificates are defined as the abstraction calculated by the analyser, the result verifier has two options. The first option is to perform the verification using the tools used to calculate the abstraction. This means that the proof of the result verifier involves the same kind of proof than the proof of the analyser itself, and the result verifier risks to be committed to a particular analysis. The second option is to translate the result of the analyser in a standard framework of program verification, but then the automation of the result verifier does not follow from the automation of the analyser.

To define a general methodology of developing result verifier, we use the second solution, and translate the result of the analyser into formulae that can be embedded into a general framework of program verification, the *deductive verification framework*, recalled in Chapter 3. The concept of *verification condition*, *i.e.*, reducing the problem to the validity of a set of first order formulae, is the basis of our result certification approach, presented in Chapter 4. Other concepts of deductive verification are re-used, such as the *memory model* of the language describes in terms of first order theories, but some complications, such as *framing conditions* for object oriented programs, are alleviated by encoding the abstraction of the program as invariants. Using verification conditions means that the computation of the result verifier is partially delegated to *automated theorem provers*. The result verification is done in two phases: first the conditions are produced given a program and the result of the analyser, then the conditions are *discharged* by automated provers. This design allows to split the overall soundness of the result verifier in two distinct problems: first the soundness of the verification conditions generator, then the soundness of the proofs of the verification conditions.

The *soundness* of the verification condition calculus and of the verification conditions *generation* is established in Chapter 5, by using a *layered* description of the verification conditions in a *intermediate verification lan-*

guage, *i.e.*, WHY3 in the experiments. First the soundness of the verification condition calculus is established with respect to the operational semantics of the language, then the verification conditions can be systematically generated, and the description of the calculus inside the intermediate language is used to prove that all the necessary conditions have been generated, and finally, the conditions are proved valid by automated provers. By using existing tools, we ensure that the result verifier can be updated easily. The description of the verification conditions calculus is independent from the condition generation, which is taken care of by the tools of the intermediate verification language, and the proving capabilities of the result verifier evolve along the improvement of automated theorem provers. Moreover, by using an established intermediate verification language, we benefit from the existing interface with a variety of provers.

The invariants encoded by numerical abstract domains belong to linear arithmetic, whose decision procedures have proved to be very efficient in a number of program verification approaches, therefore we expect the certification of numerical analyses results to pose no problem to off-the-shelf solvers. However, there are no decision procedure *tailored* for the invariants encoded by abstract domains dealing with the heap, which we expect to be challenging even for state-of-the-art solvers. As the automation of the result verifier relies on automated theorem provers, if they can not discharge the verification condition the approach can not be applied. To alleviate this problem, we introduce in Chapter 6 another verification condition calculus, designed to produce **quantifier free formulae**. This new calculus produce simpler conditions, that state-of-the-art provers are able to discharge, but that are necessarily less general. To retain some generality and avoid being dedicated to a single analysis, we describe a family of analyses using a parametrised instrumentation of the operational semantics, parametrised abstract domains, and parametrised concretisation functions. The family of analyses we describe is a restriction over analyses that deal with the heap, but contains at least the byte-code verifier and null pointer analysis we consider.

The proposed result certification methodology relies on existing tools in order to be easily implemented and upgraded. It builds trust in the result of an analyser by splitting the problem in distinct steps, therefore improving the reliability of each step improves the reliability of the whole scheme. However, using automated theorem provers gives rise to the problem of the reliability of these complex decision procedures. To alleviate this concern, we consider a particular family of automated theorem provers, the *satisfiability modulo theory* provers, recalled in Chapter 7, and apply the result certification approach to their use for proving the validity of a formula, *i.e.*, the unsatisfiability of its negation. To this end, we develop a result verifier in the proof assistant COQ, in order to provide the strongest soundness guaranties for the result verifier, and present the underlying proof system and the

architecture of the verifier in Chapter 8. One of the challenges in building a result verifier for satisfiability modulo theory solvers is that solvers do not currently agree on a particular proof system, therefore we chose to concentrate on the generality of the proof system, and the possibility to add new theories to the proof system, rather than choosing a particular solver and trying to check its results. This should allow us to built a modular result verifier, that can change its internal parts to be able to handle new theories, or to handle theories differently to benefit from new verification algorithms, without having to change the proofs and implementation of the unchanged parts. To illustrate the benefits of such an approach, in Chapter 9 we compare different verification algorithms for a particular theory, the theory of equality and uninterpreted functions, in terms of the size of the certificates and the efficiency of the verifier.

Chapter 2

Abstract interpretation

To study the certification of static analyses *in general*, without committing ourselves to a particular analysis, we chose to consider static analyses formalised in the abstract interpretation framework. The abstract interpretation theory [NNH99], introduced by P. Cousot and R. Cousot [CC76, CC77, CC79], is a general framework to build, compose, and prove the soundness of static analyses of programs. It amounts to the computation of a finite *abstract semantics* that over-approximates the possibly infinite *concrete semantics* of a program. The abstract semantics is defined over an *abstract domain* dependent on the property the analysis verifies, and a generic static analyser parametrised by these abstract domains and abstract operators can be defined.

A brief introduction to the theory of abstract interpretation is given in Section 2.1. Section 2.2 introduces a core unstructured language manipulating numerical variables which will be used in Chapter 5 to formalise numerical analyses and their certification. Section 2.3 describes some standard numerical analyses and gives a brief description of their main properties in terms of efficiency and precision. Section 2.4 presents a core object-oriented language manipulating only references. It is similar the core numerical language and will be used in the same way to formalise and certify object-oriented analyses. The object-oriented analyses selected to illustrate our approach are presented in Section 2.5.

2.1 Abstract interpretation's core mechanisms

Using the abstract interpretation framework, one can design an analysis and obtain guaranties of soundness and termination. As our goal is the certification of results of such analysers, we will concentrate first on the mechanisms that provide the soundness of an analyser. The operators necessary for termination and composition of analyses will be sketched only to account for the generality of the approach and to point out that the analyses formalised

later can actually be implemented.

2.1.1 A theory of approximation

The key principle of abstract interpretation is that any semantics can be described as a fix-point of a monotonic operator in partially ordered structures. They can therefore be compared, and new semantics can be built as abstractions of others. Abstractions are described by pairs of functions to relate two semantics, and can be manipulated as such, and composed to obtain new semantics.

Semantic domains are described by complete lattices, *i.e.*, a domain $\mathcal{D}$ equipped with a *partial order* $\sqsubseteq$, a *greatest lower bound* operator $\sqcap$, a *least upper bound* operator $\sqcup$, a *least* element $\bot$ and a *greatest* element $\top$. The semantics of a program P is given by the least fix-point of a monotonic operator F_P over $\mathcal{D}$. The semantics of the language is a set of rules [CC92b] to derive F_P from P. Kleene's theorem on fix-points in complete lattices informs us the the least fix-point $\mathit{lfp}(F_P)$ is guaranteed to exist, and that if F_P is continuous then $\mathit{lfp}(F_P) = \bigsqcap_{i \in \mathbb{N}} F_P^i(\bot)$.

Two semantic domains $\mathcal{D}$ and $\mathcal{D}^\sharp$ can be related using *Galois connections*, *i.e.*, a pair of functions (α, γ) such that:

1. $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ is monotonic, and
2. $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ is monotonic, and
3. $\forall X \in \mathcal{D}, X^\sharp \in \mathcal{D}^\sharp, \alpha(X) \sqsubseteq^\sharp X^\sharp \iff X \sqsubseteq \gamma(X^\sharp)$.

The Galois connection is used to make a correspondence between elements of the two domain. In such a configuration, the domain $\mathcal{D}$ is called the *concrete domain* and $\mathcal{D}^\sharp$ is called the *abstract domain*. The function α is referred to as the *abstraction* function, and γ is referred to as the *concretisation* function.

If an analyser computes an over-approximation in the abstract domain, *i.e.*, it compute an abstract value $X^\sharp$ such that $\alpha(\mathit{lfp}(F_P)) \sqsubseteq^\sharp X^\sharp$, then, if (α, γ) is a Galois connection, γ can be used to interpret $X^\sharp$ as an over-approximation in the concrete domain: $\mathit{lfp}(F_P) \sqsubseteq \gamma(X^\sharp)$. Moreover, the Galois connection can be used to define abstraction of operators. If F is an operator on the concrete domain, then $F^\sharp$ is said to be a *sound* approximation of F if and only if

$$\forall X \in \mathcal{D}^\sharp, (\alpha \circ F \circ \gamma)(X) \sqsubseteq^\sharp F^\sharp(X)$$

The *best* abstraction is defined as exactly $\alpha \circ F \circ \gamma$ but it may not be computed exactly in the abstract domain.

2.1.2 Building new domains

The theory of abstract interpretation defines a framework to describe abstractions and define the soundness of an abstraction *w-r-t* a concrete semantics, but the building blocks of the theory also give tools to build new abstractions. For example, the composition of Galois connections is a Galois connection, therefore relating an abstract domain to a concrete domain can be done using intermediary domains. Furthermore, if $\mathcal{D}$ is a complete lattice, then for any set $\mathcal{S}$, $\mathcal{S} \to \mathcal{D}$ can be equipped with a complete lattice structure by using operators defined by *point-wise lifting* of the operators on $\mathcal{D}$. Another important example of complete lattice is the *power-set* $(\mathcal{P}(\mathcal{S}), \subseteq, \cup, \cap, \emptyset, S)$ of any set $\mathcal{S}$.

Another possibility for creating new domains is to *combine* two existing domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$. If these domains are related to the concrete domain D by two Galois connection (α_1, γ_1) and (α_2, γ_2), then a Galois connection for the product $\mathcal{D}_2^\sharp \times \mathcal{D}_1^\sharp$ can be formed by defining $\alpha_{1\times 2}$ and $\gamma_{1\times 2}$ component-wise. However, if $F_1^\sharp$ and $F_2^\sharp$ are sound abstraction of an operator F, the operator $F_{1\times 2}^\sharp$ defined component-wise is also a sound abstraction, but it corresponds to performing the two abstraction independently at the same time: the abstraction using the two domains at once could be made more precise by using the information obtain in one domain to make the information in the other domain more precise. A more precise abstraction $G_{1\times 2}^\sharp$ can be obtained by doing the *reduced product*: a *reduction* operator ρ is defined as $\alpha_{1\times 2} \circ \gamma_{1\times 2}$, and this reduction can be used to define $G_{1\times 2}^\sharp$ as $\rho \circ F_{1\times 2}^\sharp \circ \rho$, which is a sound abstraction that is more precise than the component-wise operator $F_{1\times 2}^\sharp$.

2.1.3 Formalising a static analyser

Once the abstract domain $\mathcal{D}^\sharp$ has been defined, elements of $\mathcal{D}^\sharp$ can be interpreted as abstractions. To be able to compute in the abstract domain and to draw conclusions from the result of the computation, fix-point transfer theorems must be established. Such theorems define the conditions under which, if $F^\sharp$ is a sound abstraction of F, $\mathit{lfp}(F) \sqsubseteq \gamma(\mathit{lfp}(F^\sharp))$, *i.e.*, the fix-point computed in the abstract domain $lfp(F^\sharp)$ are approximation of fixpoint in the concrete $\mathit{lfp}(F)$, therefore it is sufficient to performer all computations in the abstract domain.

If the abstract domain $\mathcal{D}^\sharp$ contains no infinitely increasing chain, Kleenian iteration $X_{i+1} = F^\sharp(X_i)$ terminates in finite number of steps. Therefore, in such abstract domains, the fix-point can be effectively computed, and the result of the static analyser can be interpreted using fix-point transfer theorems. For domains with infinite chains, an abstract widening operator $\nabla^\sharp$ can be defined such that:

- $\forall X^\sharp, Y^\sharp, X^\sharp \sqsubseteq^\sharp (X^\sharp \nabla^\sharp Y^\sharp) \wedge Y^\sharp \sqsubseteq^\sharp (X^\sharp \nabla^\sharp Y^\sharp)$, *i.e.*, the result of the

widening is an upper bound, and

- for every inceasing chain $(X_i^\sharp)_{i\in\mathbb{N}}$, the chain

$$Y_0^\sharp = X_0^\sharp, \quad Y_{i+1}^\sharp = Y_i^\sharp \nabla^\sharp X_{i+1}^\sharp$$

is stable after a finite number of steps.

This widening operator guaranties the termination of the analyser at the cost of a loss of precision: it computes an over-approximation of the Kleenian fix-point. Precision can be regained by iterating a dual *narrowing* operator [CC92a].

To define a static analysis, one has to: i) define an abstract domain $\mathcal{D}^\sharp$ and give the Galois connection (α, γ) relating the abstract domain to the concrete semantic domain, ii) describe how to compute the sound abstraction $F_p^\sharp$ (also called *transfer function*) of the operator F_P, and finally to iii) define a widening operator $\nabla^\sharp$ such that a fix-point can be computed in finite time. The formalisation not only establishes the soundness but also describes an algorithm to compute the abstraction. However, it is a high level description of the algorithm and not a proof of the soundness of the implementation of the static analyser. To prove the soundness of the implementation of the analyser, one has to prove that the implementation of the abstract domain follows the structure of a complete lattice, in particular, the implemented operators computes a partial order, a least upper bound and so on; the iteration engine computes a fix-point; the transfer function is a sound abstraction. Pichardie *et al.* [BJPT10, CJPR05, Pic05] have demonstrated that the implementation of dataflow and polyhedral analyses can be proved in the proof assistant Coq, using constructive logic to develop and prove the algorithm and extract an executable analyser. However, such work is limited to the programming language of Coq and can not easily be applied to analysers programed in other paradigms.

2.2 Core numerical language & semantics

We present here a core unstructured language with numerical expressions, conditional jumps, procedure calls and array accesses and updates. The aim of this language is to facilitate experimentation and account succinctly for standard programming language constructs of interest.

2.2.1 Syntax

Let $\mathcal{PP}$, $\mathcal{V}ar$, $\mathcal{V}ar_{\mathcal{A}}$, $\mathcal{M}ethod$ be finite sets of program points, names for integer variables, array variables and procedure—seen as a degenerate case of methods—respectively. The set $\mathcal{V}ar$ contains distinguished elements for the parameters $\mathtt{p}_0$ and $\mathtt{p}_1$ and the result of a procedure `res`.

$$
\begin{array}{rcll}
expr & ::= & X & X \in \mathcal{V}ar \\
 & | & n & n \in \mathbb{N} \\
 & | & expr \oplus expr & \oplus \in \{+, -, \times\} \\
 & & & \\
test & ::= & expr \bowtie expr & \bowtie \in \{=, \neq, <, \leq\} \\
 & | & \texttt{not}\ test & \\
 & | & test\ \texttt{and}\ test & \\
 & | & true & \\
 & | & false & \\
 & & & \\
stmt & ::= & X := expr & X \in \mathcal{V}ar \\
 & | & \texttt{JumpIf}\ test\ p & p \in \mathcal{PP} \\
 & | & X := Y[expr] & X \in \mathcal{V}ar, Y \in \mathcal{V}ar_{\mathcal{A}} \\
 & | & X[expr] := Y & X \in \mathcal{V}ar_{\mathcal{A}}, Y \in \mathcal{V}ar \\
 & | & X := length(Y) & X \in \mathcal{V}ar, Y \in \mathcal{V}ar_{\mathcal{A}} \\
 & | & X := F(expr, expr) & F \in \mathcal{M}ethod
\end{array}
$$

Figure 2.1: Syntax of numerical programs.

The syntax of the core language, as described in Figure 2.5, contains only a limited sets of expressions—integer and Boolean—and instructions. Integer expressions are built from variables, integers and standard binary operators. Boolean expressions combine conjunctions and negations of integer comparisons. We do not have Boolean variables, as only integer expressions can be assigned to using $x := e$: Boolean expressions are used in tests only. A test `JumpIf` t p is a conditional jump to a program point p if t evaluates to *true*. Array variables are not part of integer expressions, and can only be accessed through dedicated instructions, one for accesses ($x := a[i]$) and one for updates ($a[i] = y$). A third instruction $x := length(a)$ is used to obtain the length of an array. The last instruction $x := f(e_1, e_2)$ is a procedure call, taking two arguments. For the sake of simplicity, the language does not allow an arbitrary number of arguments in procedures.

A program is a set of disjoint code sections, each code sections corresponding to a procedure. A code section is an oriented graph where nodes correspond to program points (elements of $\mathcal{PP}$) and are annotated with statements. Each program point has exactly one outgoing edge, given by the function `next_pp`, except for one point which has none (the exit point) and each point annotated with a conditional jump (`JumpIf` *test* p) which has two (one being the program point p).

Rather than modelling programs by structures built upon Abstract Syntax Trees (ASTs), the syntactic information is given directly by a flowchart: the sets of program points, variables, procedure names, and some functions

describing the control flow and the statements. The control flow is described by the function `next_pp` : $\mathcal{PP} \to \mathcal{PP}$, and the node annotations (program point statements) are accessed by a function `get_stmt` : $\mathcal{PP} \to \mathcal{S}tmt$. These sets and functions can be calculated directly on the AST of a program, but as we concentrate on unstructured languages, an analysis can not take advantage of any syntactic structure within procedures anyway.

To simplify notations, for each program point p, we write p^+ for the end of the outgoing edge ($p^+ = \texttt{next_pp}(p)$). Exit point are represented by program points p such that $p = p^+$. If p is annotated with a conditional jump, *e.g.*, `JumpIf` $t\ p'$, we note p^- the target p' of the jump. In either cases, $succ(p)$ denotes the set of all possible successors.

2.2.2 Semantics

In the seminal abstract interpretation presentation [CC77] analyses are proved correct with respect to collecting semantics. We choose to use a small-step presentation of the semantics for instructions, with a big-step reduction for procedure calls. This formalisation is called *mostly small-step* by some, and allows to use small-step derivations without relying on a call stack for intra-procedural executions.

The distinction between integer and array variables done in the languages is reflected in the memory model used in semantics states: the part of the memory containing numerical variables is represented by a mapping of variable names to value, and the part of the memory containing arrays is represented by a mapping of variables names to models of array, mappings (partial functions, noted $\rightharpoonup$) from indexes to values together with an integer representing array length. Formally, we distinguish two kinds of environments, the numerical environments $\mathcal{E}nv = \mathcal{V}ar \to \mathcal{V}al$ and the array environments $\mathcal{E}nv_{\mathcal{A}} = \mathcal{V}ar_{\mathcal{A}} \to (\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V}al))$. This simple *direct access* model does not account for more complex memory properties such as locality—*e.g.*, arrays being allocated as contiguous memory cells—or aliases, but is sufficient for the analyses we consider.

As we consider unstructured languages and we do not manipulate programs as Abstract Syntax Trees, the states of execution refers to the *current program point* of execution. Therefore, the semantic state space is a Cartesian product between the memory models and the set of program points

$$\mathcal{S}tate = \mathcal{E}nv \times \mathcal{E}nv_{\mathcal{A}} \times \mathcal{PP}.$$

While presenting the semantics, if s is of type state—hence a tuple—we use record-like notation for accessing to its different component: $s.env$ is the component of the tuple corresponding to the numerical environment, $s.ars$ is the component corresponding to the array environment, and $s.cpp$ is the component corresponding to the current program point of the execution. Likewise, if a models an array—hence of type $\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V}al)$—the

first component, corresponding to the length of the modelled array, is noted $a.len$, and the second component, the mapping between indexes and values, is noted $a.elts$. All shorthands and notation are summarised in Table 2.1. Moreover, we use a map-like notation when two functions differ only on one element: when modelling an update in an array a at index i, the array containing the new value v_2 is noted $a[i \leftarrow v_2]$. For example, in the interpretation of $x := length(a)$, the term $s.ars[a].len$ refers to the field len of the array a in the state s. To assign this value to the variable x, we simply update the store $s.env$, and this new store denoted by $s.env[x \leftarrow s.ars[a].len]$ is used to build the next state.

Shorthand	Usage
`get_stmt`(p)	Statement annotating a program point p
p^+	Program point following p in the control flow graph, shorthand for `next_pp`(p)
p^-	Destination of a conditional jump at p
$succ(p)$	Set of all possible successors of p
$\mathtt{F}_{\emptyset}$	Starting point of a procedure `F`
$\mathtt{F}_{\infty}$	Exit point of a procedure `F`
$\mathtt{p}_0, \mathtt{p}_1$	Read only variables containing the arguments during an intra-procedural execution
`res`	Variable containing the result of an intra-procedural execution
`x` *is assignable*	The variable $\mathtt{x} \notin \{\mathtt{p}_0, \mathtt{p}_1\}$
$s.env$	Local numerical environment of a state s
$s.ars$	Local array environment of a state s
$s.cpp$	Current program point of a state s
$a.len$	Length of an array a
$a.elts$	Mapping from indexes to values of an array a

Table 2.1: Shorthands used in the semantics of the core numerical language to describe a flowchart—emphasised by a `truetype` font—and a semantic state

The core numerical language uses side-effect free expressions, with neither procedure calls nor array accesses within expressions. Therefore, the semantics of expressions is simple and computing the value corresponding to an expression AST requires only to know the values of the variables, hence, the integer environment. For these reasons, we use a natural semantics—$(s, e) \Rightarrow n$ in Figure 2.2 on the following page, for a state s, and expression e and integer n—to give the semantics of expressions.

The semantics of instructions presented in Figure 2.3 on the next page is standard, and only procedure calls require more explanation. Rather

$$\text{Var}\ \frac{v \in Var}{(s, v) \Rightarrow s[v]} \qquad \text{Cst}\ \frac{n \in \mathbb{N}}{(s, n) \Rightarrow n}$$

$$\text{Binop}\ \frac{(s, e_1) \Rightarrow n_1 \quad (s, e_2) \Rightarrow n_2 \quad \oplus \in \{+, -, \times\}}{(s, e_1 \oplus e_2) \Rightarrow n_1 \oplus n_2}$$

$$\text{True}\ \frac{}{(s, true) \Rightarrow_b true} \qquad \text{False}\ \frac{}{(s, false) \Rightarrow_b false}$$

$$\text{Comp}\ \frac{(s, e_1) \Rightarrow n_1 \quad (s, e_2) \Rightarrow n_2 \quad n_1 \bowtie n_2 \quad \bowtie \in \{=, \neq, <, \leq\}}{(s, e_1 \bowtie e_2) \Rightarrow_b true}$$

$$\text{Not}\ \frac{(s, t) \Rightarrow_b b}{(s, \neg t) \Rightarrow_b \neg b} \qquad \text{And}\ \frac{(s, t_1) \Rightarrow_b b_1 \quad (s, t_2) \Rightarrow_b b_2}{(s, t_1\ and\ t2) \Rightarrow_b b_1 \wedge b_2}$$

Figure 2.2: Natural semantics of expressions and tests

$$\text{Assign}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{x} := e \quad \texttt{x}\ is\ assignable \quad (s.env, e) \Rightarrow n}{s \longrightarrow (s.env[\texttt{x} \leftarrow n]\ ,\ s.ars\ ,\ s.cpp^+))}$$

$$\text{JumpIfT}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{JumpIf}\ t\ \texttt{p} \quad (s.env, t) \Rightarrow_b true}{s \longrightarrow (s.env\ ,\ s.ars\ ,\ \texttt{p}))}$$

$$\text{JumpIfF}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{JumpIf}\ t\ \texttt{p} \quad (s.env, t) \Rightarrow_b false}{s \longrightarrow (s.env\ ,\ s.ars\ ,\ s.cpp^+)}$$

$$\text{Call}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{F}(e_0, e_1) \qquad \texttt{x}\ is\ assignable \\ (s.env, e_0) \Rightarrow n_0 \quad (s.env, e_1) \Rightarrow n_1 \quad env' = s.env[\texttt{p}_0 \leftarrow n_0][\texttt{p}_1 \leftarrow n_1] \\ init = (env'\ ,\ s.ars\ ,\ \texttt{F}_0) \qquad init \longrightarrow^* end \qquad end.cpp = \texttt{F}_\infty\end{array}}{s \longrightarrow (s.env[\texttt{x} \leftarrow end.env[\texttt{res}]]\ ,\ s.ars\ ,\ s.cpp^+))}$$

$$\text{GetArray}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{a}[e] \quad \texttt{x}\ is\ assignable \\ (s.env, e) \Rightarrow n \qquad 0 \leq n < s.ars[\texttt{a}].len\end{array}}{s \longrightarrow (s.env[\texttt{x} \leftarrow s.ars[\texttt{a}].elts[n]]\ ,\ s.ars\ ,\ s.cpp^+))}$$

$$\text{SetArray}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{a}[e_1] := e_2 \\ (s.env, e_1) \Rightarrow i \quad (s.env, e_2) \Rightarrow n \quad 0 \leq i < s.ars[\texttt{a}].len \\ elts' = s.ars[\texttt{a}].elts \quad ar = (s.ars[\texttt{a}].len\ ,\ elts'[i \leftarrow n])\end{array}}{s \longrightarrow (s.env\ ,\ s.ars[\texttt{a} \leftarrow ar]\ ,\ s.cpp^+))}$$

$$\text{Length}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{length}(\texttt{a}) \quad \texttt{x}\ is\ assignable}{s \longrightarrow (s.env[\texttt{x} \leftarrow s.ars[\texttt{a}].len]\ ,\ s.ars\ ,\ s.cpp^+))}$$

Figure 2.3: Semantics of instructions

than introducing a call-stack in the semantic states, we use a *mostly small-step* presentation: a small-step relation $\longrightarrow$ describes the evolution of the semantics of intra-procedural execution, but for procedure calls a big-step reduction of the call execution is enforced using the transitive and reflexive closure (noted $\longrightarrow^*$) of the small-step relation. To simplify the semantic rule, the parameters of procedures are fixed to $\mathtt{p_0}$ and $\mathtt{p_1}$, and all arrays are passed in argument. The result of a call is stored in the variable `res` and the integer and array environments are restored to their value before the call. As there is no dynamic dispatch, the entry point and exit point of a procedure—noted $\mathtt{F}_{\emptyset}$ and $\mathtt{F}_{\infty}$, respectively—are known in advance.

$$\text{GetArrayOoB}\ \frac{\begin{array}{c}\mathtt{get_stmt}(s.cpp) = \mathtt{GetArray\ x\ a}\ e \\ (s.env, e) \Rightarrow n \qquad n > s.ars[\mathtt{a}].len\end{array}}{s \longrightarrow OutOfBound}$$

$$\text{SetArrayOoB}\ \frac{\begin{array}{c}\mathtt{get_stmt}(s.cpp) = \mathtt{SetArray\ a}\ e_1\ e_2 \\ (s.env, e_1) \Rightarrow i \qquad i > s.ars[\mathtt{a}].len\end{array}}{s \longrightarrow OutOfBound}$$

Figure 2.4: Error conditions

Note that array bound checks are enforced using side-conditions on the semantic rules, therefore it is not possible to do out-of-bound accesses (the semantics blocks). The semantic rules GetArray and SetArray make the array bound check explicit. Array contents are then safely manipulated as maps. Figure 2.4 details the rules applied when the check fails: the semantics enters a special error state *OutOfBound*. There is no way to allocate arrays: they are supposed to be allocated, and no information is given on their size, which is only supposed to be a positive integer.

The *set of reachable states* are obtained by the reflexive, transitive closure of the relation $\rightarrowtriangle$ which enriches the semantic relation $\longrightarrow$ with states reachable from sub-calls.

$$\frac{s \longrightarrow s'}{s \rightarrowtriangle s'} \qquad \frac{\begin{array}{c}\mathtt{get_stmt}(s.cpp) = \mathtt{x} := \mathtt{F}(\mathtt{e_0}, \mathtt{e_1}) \quad x\ is\ assignable \\ (s.env, e_0) \Rightarrow n_0 \quad (s.env, e_1) \Rightarrow n_1 \\ env' = s.env[\mathtt{p_0} \leftarrow n_0][\mathtt{p_1} \leftarrow n_1] \\ s' = (env'\ ,\ s.ars\ ,\ \mathtt{F}_{\emptyset})\end{array}}{s \rightarrowtriangle s'}$$

The set of reachable states for an initial set S_0 of initial states is then defined as

$$\mathcal{R}each = \{s \mid s_0 \in S_0 \wedge s_0 \rightarrowtriangle^* s\}.$$

The semantics refers indirectly to programs through the functions describing its control-flow and statements. When a more high-level view is required, we

will allows ourselves to refer to set $\mathcal{R}each(P)$ of reachable states for a given program P.

2.3 Numerical analyses

In this section, we aim to give a high-level summary of the abundant literature on numerical analyses to illustrate what kind of properties we will and will not consider. We do not propose any improvement but introduce notations and formalisation reused in Chapter 5.

2.3.1 Interval analyses

Interval arithmetic was introduced by R. E. Moore in the 1960's [Moo62] to provide rigorous bounds on round-off errors during computation, starting a whole new field in scientific computing. Cousot and Cousot adapted the classical interval operators to the concepts of abstract interpretation in their presentation [CC76], introducing the interval abstract domain.

Abstract domain. Let $\mathbb{D}$ be a numerical domain : $\mathbb{D} \in \{\mathbb{Z}, \mathbb{R}, \mathbb{Q}\}$.

The lattice $(\mathbb{D}^{int}, \sqsubseteq_{\mathbb{D}}^{int}, \cup_{\mathbb{D}}^{int}, \cap_{\mathbb{D}}^{int}, \bot_{\mathbb{D}}^{int}, \top_{\mathbb{D}}^{int})$ of intervals is defined as follows.

- $\mathbb{D}^{int}$ is the set of all intervals: an interval is a pair $[a, b] \in (\mathbb{D} \cup \{-\infty\}) \times (\mathbb{D} \cup \{+\infty\})$ such that $a \leq b$, or the empty interval, noted $\bot_{\mathbb{D}}^{int}$. The interval $[-\infty, +\infty]$ is written $\top_{\mathbb{D}}^{int}$.
- The partial order $\sqsubseteq_{\mathbb{D}}^{int}$ on $\mathbb{D}^{int}$ is defined by
  - $[a, b] \sqsubseteq_{\mathbb{D}}^{int} [a', b']$ if and only if $a' \leq a$ and $b \leq b'$,
  - and $\bot_{\mathbb{D}}^{int}$ is the least element.
- Suppose *min* and *max* are lifted to $\mathbb{D} \cup \{+\infty, -\infty\}$.

$$X^\sharp \cup_{\mathbb{D}}^{int} Y^\sharp = \begin{cases} [min(a, a'), max(b, b')] & \textit{if } X^\sharp = [a, b] \textit{ and } Y^\sharp = [a', b'] \\ X^\sharp & \textit{if } Y^\sharp = \bot_{\mathbb{D}}^{int} \\ Y^\sharp & \textit{if } X^\sharp = \bot_{\mathbb{D}}^{int} \end{cases}$$

$$X^\sharp \cap_{\mathbb{D}}^{int} Y^\sharp = \begin{cases} [max(a, a'), min(b, b')] & \textit{if } X^\sharp = [a, b] \textit{ and } Y^\sharp = [a', b'] \\ & \quad \textit{and } max(a, a') \leq min(b, b') \\ \bot_{\mathbb{D}}^{int} & \textit{otherwise} \end{cases}$$

Note that the intersection is *exact*—no information is lost—while the union is not, therefore, when performing a join (when analysing nodes with several predecessors) when doing forward analysis, some information might be lost.

Intervals are abstraction of values, and the abstraction of a set of states is a lift of the interval domain: for each program point, one interval is given for each variable.

$$\mathcal{State}^\sharp = \mathcal{PP} \to \mathcal{Var} \to \mathbb{D}^{int}$$

Concretisation. The concretisation function γ of the abstraction of a program $P^\sharp$ is a lift of the concretisation of values $\gamma^{int}_{\mathcal{Val}}$.

$$\begin{array}{ll} \gamma^{int}_{\mathcal{Val}}([a,b]) & = \{x \in \mathbb{D} \mid a \leq x \wedge x \leq b\} \\ \gamma^{int}_{\mathcal{Val}}(\bot^{int}_{\mathbb{D}}) & = \emptyset \\ \gamma(P^\sharp) & = \left\{(e,a,p) \in \mathcal{State} \mid \forall v \in \mathcal{Var}, e(v) \in \gamma^{int}_{\mathcal{Val}}(P^\sharp(p,v))\right\} \end{array}$$

Widening and narrowing. To achieve termination we need operators to over-approximate limits of infinitely increasing and decreasing chains. One possibility, informally described, is to set to $\pm\infty$ the bounds that increase when calculating the abstraction of a variable. The formal definition of the widening operator and possibility to refine its results can be found in the formalisation by P. Cousot of the interval analysis in the abstract interpretation theory [CC77].

2.3.2 Polyhedron analyses

Interval analyses are concerned with discovering *bounds* on variable: they can not used to infer invariants establishing *relations* between variables. The simple example given in Listing 2.1 illustrates the need for relational domains: if the length of the array is not known, then an interval analysis will only be able to prove that X belongs to the interval $[-1, +\infty[$, therefore it will not be able to prove that X < length(A) and the second instruction will raise an alarm. However, if the analyser can prove relational invariant, (*e.g.*, $X < length(A)$, $X = length(A) - 1$, etc.), then proving that the array access makes no array out of bound error is possible.

```
X := length(A) - 1
Y := A[X]
```

Listing 2.1: Example of a program that requires relational information

In the early 70's, the problem of discovering affine relationships among variables was explored by Wegbreit [Weg74] and Karr [Kar76]. Cousot and Halbwachs [CH78] used the framework of abstract interpretation to guarantee termination while searching for linear inequalities, defining an abstract domain of *convex polyhedrons*. The algorithms used for operators on the polyhedron abstract domain suppose $\mathbb{D} \in \{\mathbb{R}, \mathbb{Q}\}$. In fact, the problem of finding integer solutions for linear inequalities is NP-complete whereas there is a polynomial algorithm for finding rational solutions. When dealing with

integer variables, a common (sound) solution is to abstract set of points with integer coordinates by a rational polyhedron.

Abstract domain. Let $\{X_i \mid 0 \leq i \leq n\}$ be the set of variables. A valuation of all variables can be represented by a vertex in $\mathbb{D}^n$. A *linear constraint* is an inequality $\sum_{0 \leq i \leq n} a_i X_i \leq b$. The set of solutions of such an inequality (if non-empty) defines a *closed half space* in $\mathbb{D}^n$.

A set $S \subseteq \mathbb{D}^n$ is *convex* if and only if

$$\forall x, y \in S^2, \forall \lambda \in [0,1], \lambda x + (1-\lambda) y \in S$$

For example, a closed half space is a convex set, the intersection of two convex set is a convex set, but the union of two convex sets is not. The set of solutions of a system of linear constraints

$$\left\{ \sum_{0 \leq i \leq n} a_{ij} X_i \leq b_j \;\middle|\; 0 \leq j \leq m \right\}$$

is a convex set, represented by the intersection of the closed half space: a convex (possibly unbounded) polyhedron. Such a polyhedron can be represented by a matrix $A \in \mathbb{D}^{n \times m}$ and a vertex $B \in \mathbb{D}^m$.

A dual representation of convex polyhedrons can be given in terms of a set of vertexes $V = V_1, \ldots, V_k$ and a set of rays $R = R_1, \ldots, R_l$. The vertexes are extremal points in the polyhedron, and the rays are unbounded direction included in the polyhedron. Any points in the polyhedron P can be expressed as a combination of vertexes and rays :

$$\forall X \in P, \exists (\lambda_i)_{1 \leq i \leq k} \in [0,1]^k, (\nu_i)_{1 \leq i \leq l} \in \mathbb{D}^{+l}, \\ \sum_{1 \leq i \leq k} \lambda_i = 1, X = \sum_{1 \leq i \leq k} \lambda_i V_i + \sum_{1 \leq i \leq l} \nu_i R_i$$

Polyhedron are abstraction of environments, and we can define a concretisation function for each representation. An environment is a function from variables to values, and can therefore be represented by a vertex $\vec{E} \in \mathbb{D}^n$.

$$\gamma^{poly}_{\mathcal{E}nv}(A, \vec{B}) = \left\{ \vec{E} \in \mathcal{E}nv \;\middle|\; A\vec{E} \leq \vec{B} \right\}$$

$$\gamma^{poly}_{\mathcal{E}nv}(V, R) = \left\{ \sum_{1 \leq i \leq k} \lambda_i \vec{V_i} + \sum_{1 \leq i \leq l} \nu_i \vec{R_i} \;\middle|\; \lambda_i \in [0,1]^k, \nu_i \in \mathbb{D}^{+l}, \sum_{1 \leq i \leq k} \lambda_i = 1 \right\}$$

As it is the case for the interval analysis, the abstraction of a program and the associated concretisation function are lifted from the base domain (polyhedron) and its concretisation.

$$\begin{aligned} \mathcal{State}^\sharp &= \mathcal{PP} \to \mathbb{D}^{poly} \\ \gamma(P^\sharp) &= \left\{ (e, a, p) \in \mathcal{State} \;\middle|\; e \in \gamma^{poly}_{\mathcal{E}nv}(P^\sharp(p)) \right\} \end{aligned}$$

The widening operator proposed by Cousot and Halbwachs is comparable to the one used in interval analysis: if a polyhedron keeps on increasing in one direction, the whole direction is included in the abstraction as an over-approximation.

Efficiency. We mentioned two different representation because some operators and transfer functions are more easily defined on linear constraints, while others are easier to define on vertexes and rays. The size of these two representations is, in practice, exponential in terms of *variables*,[1] and algorithms to switch between representations have worst-case exponential time and memory cost [Che68]. Therefore the polyhedron abstract domain, if very precise, is very costly. Section 2.3.3 describe the Octagon domain, that defines an analysis less precise than the polyhedron analysis, but that can still find relational invariants, and can be computed much more efficiently.

Example. In the program described in Listing 2.2 on the following page, the `search` procedure implements a binary search. To prove that the procedure raise no *array out of bound* exception, a static analyser must prove that the variable i must be positive and strictly less than the length of the array, $0 \leq i \leq t.\texttt{length}$.

Using the interproc analyser [Jea] in its polyhedron settings, we can obtain the annotated invariant. Not all the constraints of the invariant are useful, but using the last 5 linear inequalities, we can prove that

$$0 \leq 2i + 2 \leq a + b \leq t.\texttt{length} - 1$$

hence that no *array out of bound* exception is raised.

2.3.3 Octagon analysis

The polyhedron domain allows inference of useful relational information, whereas the interval domain provides non-relational information. However, the exponential cost of inference of polyhedrons makes it difficult to use in practice. To infer relational information while retaining part of the efficiency

[1] Remark that the *complexity* of the polyhedron abstract domain is different from the complexity of solving a set of linear inequalities in $\mathbb{Q}$. The former should be in terms of a metric on the program, such as the number of variable, and the number of inequalities is not bounded, and the latter is in terms of the size of the problem, and is polynomial.

```
proc search (v : int, t : int array) returns (i : int)
var a : int , b : int , arr : int;
begin
   assume t.length > 0;
   a = 0;
   b = t.length - 1;
   arr = 0;
   while (arr==0) and (0 < b - a) do
      i = a + (b-a)/ 2;
      (* (L9 C27)
          [|arr=0; -a-b+2i+2≥0; -a+b-1≥0;
             -b+t.length-1≥0; i≥0; a≥0; a+b-2i+2≥0|] *)
      if t[i] == v then
          arr = 1;
      else
          if v > t[i] then
             a = i+1;
          else
             b = i-1;
          endif ;
      endif ;
   done ;
end
```

Listing 2.2: Binary search

of simple analyses like the interval analysis, abstract domains based on particular subset of linear constraints with more efficient algorithms are needed. A. Miné introduced *weakly relational domains* [Min04, Min06]: numerical abstract domain based on subset of linear constraint for which algorithms with good complexity can be devised. They are based on known data structures and algorithms, which were adapted and expended to fit the abstract interpretation framework.

Octagon abstract domain. An *octagonal constraint* is an inequalities $\pm X_i \pm X_j \leq c$, and an *octagon* is a conjunction of octagonal constraints—the name octagon reflect the geometrical representation of these abstraction in dimension 2. Octagons can be encoded by *Difference Bound Matrices* (DBMs, square matrices of size $n \times n$), coefficients in the matrix standing for octagonal constraints.

For this presentation, abstraction of programs and concretisation can be easily adapted from the concretisation of polyhedron based on linear constraints.

$$\gamma_{oct}(O) = \{(e, a, p) \mid \forall(\pm X_i \pm X_j \leq c) \in O, \pm e(X_i) \pm e(X_j) \leq c\}$$

Efficiency. A cubic closure algorithm is needed to obtain a normal form and attain optimality, therefore an octagon based analysis can only achieve

$\mathcal{O}(n^3)$ worst-case time complexity, where n is the number of variables. In practice, efficient algorithms such as *incremental closure* and the use of closure preserving operators make the octagon abstract domain very efficient. All operations on the DBM encoding can be performed with a $\mathcal{O}(n^2)$ worst-case memory complexity, and in practice, the use of a *sparse* matrix representation can improve memory consumption, at the cost of time efficiency.

For a complete description of all the operators needed to build a full analyser and more details on algorithms and implementation consideration, we refer the reader to the dissertation of A. Miné [Min04].

2.3.4 Conclusion

The three abstract domains we described have different precision and different computational properties, but they can all be represented by linear constraints. There are lots of other numerical domains, either relational or not. Some can be embedded in the polyhedron domain, other are dedicated to completely different properties. We only presented the analyses that were used in our experiments.

2.4 Core object-oriented language & semantics

The core object-oriented language we use is based on the same premises than the core numerical language presented in Section 2.2: it is unstructured—rely on program point for a description of the control flow—and uses the simplest expressions and instruction set needed to illustrate and evaluate our approach with the respect to the object-oriented paradigm.

The goal is to illustrate some particularity of bytecode while retaining simplicity of a core language. It therefore concentrate on field dereferencing and dynamic method call, therefore modelling aliases through the heap and dynamic dispatch. It does not contain expressions on integers for example, that a simple exposition of an analyses of heap ignores anyway.

2.4.1 Syntax

The syntactic domains of the core object-oriented language differ from the numerical language. We have fields instead of arrays, and procedures are replaced by methods and classes. Therefore we define the finite sets $\mathcal{PP}$, $\mathcal{V}ar$, $\mathcal{M}ethod$, $\mathcal{C}lass$ and $\mathcal{F}$ for program points, variable names, classes, method names and field names. The set $\mathcal{V}ar$ contains distinguished elements for the `this` pointer, the parameters $\mathtt{p}_0$ and $\mathtt{p}_1$ and the result of a method `res`. Figure 2.5 details all statements and expressions.

$$
\begin{array}{rcll}
\mathcal{V}ar & ::= & \texttt{this} \mid \texttt{p}_0 \mid \texttt{p}_1 \mid \texttt{res} \mid \ldots & \\
\\
\mathcal{E}xpr & ::= & X & X \in \mathcal{V}ar \\
 & \mid & \mathit{Null} & \\
\\
\mathcal{S}tmt & ::= & X := \mathit{expr} & \\
 & \mid & \texttt{IfNull}(X, p) & X \in \mathcal{V}ar, p \in \mathcal{PP} \\
 & \mid & X := Y.F & X, Y \in \mathcal{V}ar^2, F \in \mathcal{F} \\
 & \mid & X.F := Y & X, Y \in \mathcal{V}ar^2, F \in \mathcal{F} \\
 & \mid & X := Y.C.M(X_0, X_1) & X, Y, X_0, X_1 \in \mathcal{V}ar^4, \\
 & & & \quad C \in \mathcal{C}lass, M \in \mathcal{M}ethod \\
 & \mid & X := \texttt{new}(C) & X \in \mathcal{V}ar, C \in \mathcal{C}lass \\
 & \mid & \texttt{skip} &
\end{array}
$$

Figure 2.5: Syntax of object-oriented programs.

Expressions. Values are defined to be objects, therefore expressions are restricted to variables or the defaults value *Null*. The only possibility to create a new value is the $X := \mathit{new}\ C$ instruction, which allocate a new object on the heap and assign a pointer to this object to a variable.

Statements. The object-oriented language has the same base instructions as the numerical language (assignment, test and call), but their mechanics have changed. The assignment of expressions is still there, but we do not have Boolean expressions. Instead, the language has a *IfNull* test: it takes a pointer (the content of a variable) and makes the jump if the pointer is *Null*.

Paths are not included in the language, to assign an object referred to by a path such as $X.F.G.H$ we need to use intermediary variables and use the *putfield* instruction. $Y := X.F.G.H$ is decomposed in a sequence of *putfield*s: $Y_1 := X.F; Y_2 := Y_1.G; Y := Y_2.H$.

The $X := \mathit{new}\ C$ instruction allocates a new object of class C in the heap. It does not execute any constructor, but initialise all fields with the *Null* value.

The method call $X := Y.C.M(X_0, X_1)$ takes several arguments: the variable X that will store the result, the variable Y containing the caller, the name of the method M, the arguments X_0 and X_1 (which are variables, not expressions), and the additional arguments C, a class in which the method M is defined (as in bytecode instructions). The call is a dynamic call: a method can have multiple implementations as subclasses can redefine the methods of their ancestors, and a *lookup* algorithm determines at runtime the implementation to use. The argument C gives a class name which im-

plements M, and the lookup can only return implementations that belongs to subclasses of C.

Programs. The model of programs is similar to the one used in the numerical language, with additional information to account for methods and a class hierarchy. A method $(\mathtt{c}, \mathtt{m}) \in \mathcal{Class} \times \mathcal{Method}$ is identified by its name $\mathtt{m}$ and its defining class $\mathtt{c}$. Its entry point (written $(\mathtt{c.m})_0$) and its exit point (written $(\mathtt{c.m})_\infty$) are given by the mapping $\mathtt{sig} \in \mathcal{Class} \times \mathcal{Method} \to (\mathcal{PP} \times \mathcal{PP})_\perp$. The mapping $\mathtt{sig}$ is effectively a partial function, but contrary to arrays, rather than being called only when it is defined thanks to array bound checks, it returns a special value $\perp$ if it is not defined. The code is described via the same two functions: $\mathtt{get_stmt} \in \mathcal{PP} \to \mathcal{Stmt}$ returns the statement at program point; $\mathtt{next_pp} \in \mathcal{PP} \to \mathcal{PP}$ returns the successor of a program point. The only exception is the conditional statement $\mathtt{IfNull}(e, p)$ whose successor is p if e is null. The class hierarchy is represented by a partial order $\preceq$ over classes. Each class defines a set of fields. We write $f \in c$ for a field that is either defined or inherited by c, *i.e.,* defined by a super-class of c. The `lookup` function models virtual method dispatch and is defined if a matching method is found by walking up the class hierarchy:

$$\mathtt{lookup} : \mathcal{Class} \to (\mathcal{Class} \times \mathcal{Method}) \to \mathcal{Class}_\perp$$

We use the same simplified notations p^+, p^- and $succ(p)$, and all shorthands and notations are summarised in Table 2.2 on the next page.

2.4.2 Semantics

The memory model used in the semantics of the core object-oriented language differs from the one used for the numerical language semantics, to account for the heap and the absence of arrays.

The semantic domains are built upon an infinite set of location $\mathcal{L}$; values are either a location or null; an environment is a mapping from variables to values; an object is a pair made of a class and a mapping from fields to value ; the heap is a partial mapping from locations to objects. A state is a tuple of $\mathcal{Env} \times \mathcal{Heap} \times \mathcal{PP}$. Again, for accessing states we use a record-like notation: if a logical variable s refers to a state (e, h, p), then $s.env$ is the environment e, $s.hp$ is the heap h, and $s.cpp$ is the current program point p. We add a set of error states $\mathcal{Err}$ for null pointer dereferencing and calls to undefined methods or lookup failure.

$$\mathcal{Val} = \mathcal{L} \cup \{null\} \qquad \mathcal{Env} = \mathcal{Var} \to \mathcal{Val} \qquad \mathcal{Obj} = \mathcal{Class} \times (\mathcal{F} \to \mathcal{Val})$$
$$\mathcal{Heap} = \mathcal{L} \to \mathcal{Obj}_\perp \qquad \mathcal{State} = \mathcal{Env} \times \mathcal{Heap} \times \mathcal{PP}$$
$$\mathcal{Err} = \{NullPointer, LookupFail\}$$

Shorthand	Usage
$\texttt{get_stmt}(p)$	Statement annotating a program point p
p^+	Program point following p in the control flow graph, shorthand for $\texttt{next_pp}(p)$
p^-	Destination of a conditional jump at p
$succ(p)$	Set of all possible successors of p
$\texttt{sig}(c, m)$	Signature of a method m in a class c, can be either a pair of program pointsor $\perp$ if the method is not implemented
$(c.m)_0$	Starting point of of a method m in a class c
$(c.m)_\infty$	Exit point of of a method m in a class c
$c_1 \preceq c_2$	The class c_1 is defined as a subclass of c_2
$f \in c$	The field f is defined or inherited by the class c
$\texttt{lookup}(c)(c_0, m)$	Result of the lookup of a method m starting at c and finishing atc_0, can be either a class or $\perp$ if the lookup failed
$\texttt{p}_0, \texttt{p}_1$	Read only variables containing the arguments during an intra-procedural execution
$\texttt{res}$	Variable containing the result of an intra-procedural execution
$\texttt{x}$ *is assignable*	The variable $\texttt{x} \notin \{\texttt{p}_0, \texttt{p}_1, \texttt{this}\}$
$s.env$	Local environment of a state s
$s.hp$	Heap of a state s
$s.cpp$	Current program point of a state s
$\lambda f.null$	Uninitialised object
$\lambda v.null$	Uninitialised environment

Table 2.2: Shorthands used in the semantics of the core object-oriented language to describe a flowchart—emphasised by a `truetype` font—and a semantic state

Expressions are restricted to be either variables or `null`, so their semantics given below is simple.

$$\frac{}{(\sigma, \texttt{null}) \Rightarrow null} \qquad \frac{}{(\sigma, v) \Rightarrow \sigma(v)}$$

The semantics of a program is presented in Figure 2.6 and Figure 2.7 on page 29 using the same map-like notation—$m[k]$ for accesses and $m[k \leftarrow v]$ for updates—we used in Section 2.2. Again, we use a *mostly small-step* presentation of the semantics, defining inductively a relation $\longrightarrow$ between states and its transitive closure $\longrightarrow^*$. The environment of initial state while executing a method call is built upon the environment at the call point. The variable `this` is assigned with the callee, and the two arguments of the call are assigned to the parameters $\texttt{p}_0$ and $\texttt{p}_1$. No instruction allows explicit

assignment to the variables $\texttt{this}$, $\texttt{p}_0$ and $\texttt{p}_1$, which are effectively *read-only* variables. The fields of newly created objects are initialised with the *null* value.

Most of the semantics of instruction is standard. The *skip* instruction, the assignment and the conditional jump behave as expected. The allocation of an object initialise all fields to *null*, and select a location l that is unallocated in the heap ($s.hp[l] = \bot$). Accesses and updates in the heap are decomposed in two steps: first the location is fetched in the environment, then the object is fetched in the heap. An object of class c is defined as a pair (c, o) where o is a mapping from fields to value, therefore the class of an object appears in the rule for heap accesses even though it is not used.

The relation $s \rhd_{y.c_0.m(a_0,a_1)} (s', p_{end})$ defined by the rule *SCall* in Figure 2.6 on the next page is a shorthand for the definition of the initial state of a call to the method $c_0.m$: a method with name m defined in the class c_0. As we use a dynamic dispatch, the implementation of the method to execute is selected by the lookup algorithm. The relation $\rhd_{y.c_0.m(a_0,a_1)}$ also defines the exit point $p_{\texttt{end}}$ of the implementation to execute. The initial environment of a method call only contains value for the variables $\texttt{this}$, $\texttt{p}_0$ and $\texttt{p}_1$—respectively, the caller and the values of the two arguments.

Figure 2.7 on the following page describes the semantics of programs "which go wrong". The semantics is blocking with respect to ill-formed programs (assignment to the variables $\texttt{this}$, $\texttt{p}_0$ and $\texttt{p}_1$), but programs leading to null pointer dereferencing or *method not found* lead to special error states (*NullPointer* and *LookupFail*).

The *set of reachable states* are obtained by the reflexive, transitive closure of the relation $\rightarrowtriangle$ which enriches the semantic relation $\longrightarrow$ with states reachable from sub-calls in similar way as for the numerical core language, but adapted to the dynamic dispatch mechanism.

$$\frac{s \longrightarrow s'}{s \rightarrowtriangle s'} \qquad \frac{\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x \textit{ is assignable} \qquad s \rhd_{y.c_0.m(a_0,a_1)} (s', p_{end})}{s \rightarrowtriangle s'}$$

The set of reachable states for an initial set S_0 of initial states is then defined as

$$\mathcal{Reach} = \{s \mid s_0 \in S_0 \wedge s_0 \rightarrowtriangle^* s\}.$$

As for the numerical core language, the semantics refers indirectly to programs through the functions describing its control-flow and statements. When a more high-level view is required, we will allows ourselves to refer to set $\mathcal{Reach}(P)$ of reachable states for a given program P.

Definition 2.4.1 formalises the well-formedness properties of semantic states that are preserved by the semantic rules.

$$\text{Skip}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{skip}}{s \longrightarrow (s.env, s.hp, s.cpp^{+})}$$

$$\text{Assign}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{e} \quad x \textit{ is assignable} \quad (s.env, e) \Rightarrow v}{s \longrightarrow (s.env[x \leftarrow v]\ ,\ s.hp\ ,\ s.cpp^{+})}$$

$$\text{JumpNull}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{IfNull}(\texttt{t}, \texttt{p}') \quad (s.env, t) \Rightarrow null}{s \longrightarrow (s.env\ ,\ s.hp\ ,\ p')}$$

$$\text{JumpLoc}\ \frac{\texttt{get_stmt}(s.cpp) = \texttt{IfNull}(\texttt{t}, \texttt{p}') \quad (s.env, t) \Rightarrow l \quad l \neq null}{s \longrightarrow (s.env\ ,\ s.hp\ ,\ s.cpp^{+})}$$

$$\text{New}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = x := \texttt{new c} \quad x \textit{ is assignable} \\ s.hp[l] = \bot \qquad o = \lambda f.null\end{array}}{s \longrightarrow (s.env[x \leftarrow l]\ ,\ s.hp[l \leftarrow (c, o)]\ ,\ s.cpp^{+})}$$

$$\text{Getfield}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{y.f} \qquad x \textit{ is assignable} \\ s.env[y] = l \quad l \neq null \quad s.hp[l] = (c, o) \quad o[f] = v\end{array}}{s \longrightarrow (s.env[x \leftarrow v]\ ,\ s.hp\ ,\ s.cpp^{+})}$$

$$\text{Putfield}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x.f} := \texttt{y} \qquad s.env[x] = l \\ l \neq null \quad s.hp[l] = (c, o) \quad v' = s.env[y] \quad o' = o[f \leftarrow v']\end{array}}{s \longrightarrow (s.env\ ,\ s.hp[l \leftarrow (c, o')]\ ,\ s.cpp^{+})}$$

$$\text{Call}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \qquad x \textit{ is assignable} \\ s \rhd_{y.c_0.m(a_0,a_1)} (init, p_{end}) \quad init \longrightarrow^{*} end \quad end.cpp = p_{end}\end{array}}{s \longrightarrow (s.env[x \leftarrow end.env[\texttt{res}]]\ ,\ end.hp\ ,\ s.cpp^{+})}$$

$$\text{SCall}\ \frac{\begin{array}{c}s.env[y] = l \quad l \neq null \quad (s.env, a_0) \Rightarrow v_0 \quad (s.env, a_1) \Rightarrow v_1 \\ s.hp[l] = (c, o) \quad \texttt{lookup}(c)(c_0, m) = c' \quad \texttt{sig}(c', m) = (p_{beg}, p_{end}) \\ env' = (\lambda x.null)[\texttt{this} \leftarrow l][\texttt{p}_0 \leftarrow v_0][\texttt{p}_1 \leftarrow v_1]\end{array}}{s \rhd_{y.c_0.m(a_0,a_1)} ((env'\ ,\ s.hp\ ,\ p_{beg}), p_{end})}$$

Figure 2.6: Semantics of the object-oriented core language

$$\text{GetfieldNullP}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{y.f} \quad x \textit{ is assignable} \\ s.env[y] = null\end{array}}{s \rightsquigarrow NullPointer}$$

$$\text{PutfieldNullP}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x.f} := \texttt{y} \quad x \textit{ is assignable} \\ s.env[x] = null\end{array}}{s \rightsquigarrow NullPointer}$$

$$\text{CallNullP}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x \textit{ is assignable} \\ s.env[y] = null\end{array}}{s \rightsquigarrow NullPointer}$$

$$\text{LookupFail}\ \frac{\begin{array}{c}\texttt{get_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x \textit{ is assignable} \\ s.env[y] = l \qquad l \neq null \qquad s.hp[l] = (c, o) \\ \texttt{lookup}(c)(c_0, m) = \bot\end{array}}{s \rightsquigarrow LookupFail}$$

$$\ldots$$

Figure 2.7: Limited subset of the error conditions

Definition 2.4.1 (Well-formedness)**.** A semantic state $s = (e, h, p) \in \mathcal{State}$ is well-formed ($Wf(s)$) if and only if there are no dangling pointers.

$$\forall l.h(l) = (c, o) \Rightarrow \forall f, l'.o(f) = l' \Rightarrow l' \in dom(h)$$
$$\forall x, l.e(x) = l \Rightarrow l \in dom(h)$$

The semantics also enforces certain properties of consecutive states that are summarised by the notion of *compatible states*, defined as follows.

Definition 2.4.2 (Compatibility)**.** A state $s = (e, h, p)$ and a state $s' = (e', h', p')$ are compatible ($Compat(s, s')$) with the semantic relation if and only if

1. The resulting heap h' defines more locations than the initial heap h'
2. The class of objects is invariant
3. Arguments of method calls (including the variable `this`) are immutable

$$\forall l.l \in dom(h) \Rightarrow l \in dom(h') \quad (1)$$
$$\forall c, c', l, o, o'.h(l) = (c, o) \Rightarrow h'(l) = (c, o') \Rightarrow c = c' \quad (2)$$
$$e(\texttt{this}) = e'(\texttt{this}) \wedge e(\texttt{p}_0) = e'(\texttt{p}_0) \wedge e(\texttt{p}_1) = e'(\texttt{p}_1) \quad (3)$$

Proposition 2.4.1 formally states that the semantics preserves the well-formedness conditions (Definition 2.4.1) and that consecutive states are compatible (Definition 2.4.2).

Proposition 2.4.1. *Given two consecutive states s and s' ($s \rightarrow s'$), if s is well-formed $Wf(s)$ then s' is well-formed ($Wf(s')$) and compatible with s ($Compat(s, s')$).*

2.5 Object-oriented analyses

Our first concern while choosing an analysis to certify is to access the feasibility of our approach, but we still want to consider an analysis relevant to object-oriented languages. The BCV is full of simple properties for which analysis are well understood. However, for this very reason, mechanical proofs of such analysers already exist [Pus99, KN03]. If we want to provide an alternative to complex mechanised proofs, we need to consider analyses on which they are hard to provide. We chose to concentrate on Null-pointer analyses, and describe a whole family of analyses using one formalisation by using the parametric instrumentation of semantics presented in Section 2.4.

2.5.1 Simple analyser : ByteCode Verifier

We give a brief description of the Java Virtual Machine and enlighten some features of interest for our approach, then formalise an aspect of Bytecode Verification to which our result certification approach will be applied in Chapter 6. More information on Java bytecode verification, its algorithms and their formalisations [FM98] can be found in a survey by X. Leroy [Ler03].

Bytecode language. The Java Virtual Machine (JVM) [LY99] is an abstract machine using stacks and registers. Programs are written in an unstructured, typed, instruction-based language (the *Java bytecode*), and instructions pops arguments off and push their results on a stack. Local variables are represented by a given set of registers accessed through *load* and *store* instructions that pops off and push on the stack the content of registers.

The Java bytecode is unstructured in the sense that control is handled by *goto*-like instructions (conditional or not). But a bytecode program is composed of delimited methods with entry and exit points, and jump can only be *intra-procedural* (approximately, see *subroutines*). Methods are called with dedicated instructions such as *invokestatic* and *invokevirtual* and each method has an *activation record* containing its stack and register, preserved across method calls.

As mentioned before, most instructions of the JVM bytecode are typed. For example, the arithmetic addition instruction `iadd` requires that the top two elements on the stack be integers, pop them off, and push an integer on the stack. Objects fields are manipulated through `getfield` and `putfield` *typed* instructions. More precisely, a `getfield` $C.f.\tau$ instruction requires a pointer to an object of class C or one of its subclass at the top of the stack, pops it off, and pushes on a value of type τ (the value contained in field f). Similarly, method calls are of the form $y.C.m(\ldots)$ and the lookup algorithm for dynamic calls will fail if the dynamic class of y is not a subclass of C.

Bytecode Verifier. The JVM offers no guaranty of proper execution if the following requirements are not met:

- No stack underflow or overflow: No instruction tries to pop a value from an empty stack, nor does it try to add a value on a stack of the maximum size specified on the entry point of its method.
- Code containment: The program counter always points to the beginning of a valid instruction inside the method.
- Register initialisation: a load from a register always follow a store on that register.
- Object initialisation: After creation of an object and before it is used, one of the initialisation method of its class must be invoked.
- Type correctness: The arguments of an instruction are always of the type expected by the instruction.

A *defensive JVM approach* consists in checking these requirements dynamically, but it is expensive and slows down the execution. Verifying these conditions or part of them *statically* on the code of a program at loading

time speeds up the execution, as bytecode passing the verification can be executed with fewer dynamic checks.

Type correctness. Most conditions can not be interpreted in our settings. We consider a stack free, register free language and suppose code containment holds. Moreover, we only model dynamic calls, and do not make a distinction between the constructors and the other methods, therefore object initialisation can not be modelled directly. The main property we are interested in is type correctness, with the exception of basic types.

For any program, the subtyping relation used in the JVM, written $<:$, respects the `extends` relation defined in the program, lifts it to arrays of arbitrary dimension and adds the class `null` for null pointers (subtype of all classes), the class `Object` ancestor of all classes and arrays, and the class $\top$, ancestor of the class `Object` and all basic types (integers, floating point real numbers, *etc.*). The set of types ordered by $<:$ is a semilattice (any pair of types has a least upper bound), and $<:$ is well founded (there is no infinite strictly increasing chain of types).

2.5.2 Null pointer analysis

In object-oriented language, a programmer needs to distinguish proper objects from the special null value or base values. Modern languages such as C# and Java do not provide type information of that sort, *i.e.*, null is a correct value of any class/type. Other languages like SPEC# do make such a distinction, and the declaration of a field of non-null type ensures that every read access yield a non-null value and every update requires a non-null value. The key property is that a object/record under construction cannot be accessed until fully constructed. But languages like C# and JAVA gives access to partially initialised object through the variable `this`, and cannot enforce this kind of properties in general.

Fähndrich and Leino [FL03] propose a type system allowing annotation of fields with non-null types, and a type checking algorithm to verify an annotated object-oriented program, making sure that annotations are consistent with the program's behaviour. Contrary to BCV, this is about enriching the language with a new type system. Whereas the BCV is integrated to the JVM, non-null types are not enforced by the BCV, and in fact not all programs with no non-null error can be typed. However, proving the absence of null dereferencing error using non-null type systems makes possible the use of less dynamic checks in a defencive programming setting, and as usual, provides faster execution.

Raw types. The first conceptual step in creating a non-null type system, is to introduce for every declared class T, a distinguished type T^- for non-null references of type T. To avoid confusion, let T^+ be the type of references

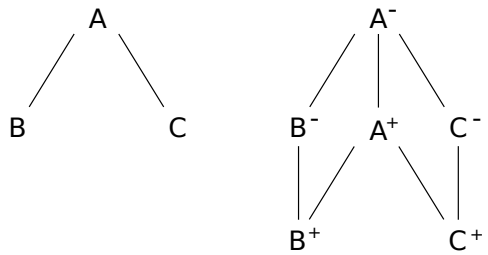


Figure 2.8: Non-null types are lifted to a lattice (on the right) following the class hierarchy (on the left)

of class T that can be null. Annotating a program with non-null annotations now amounts to using the type T^- and T^+ whenever a type is expected, *e.g.*, in method signatures or variable declarations. This distinction between possibly null and non-null values can be lifted to the class hierarchy, as illustrated in Figure 2.8. If S is a subclass of T, then $S^+ \preceq T^+$ and $S^- \preceq T^-$. Naturally, all variables of type T^- can be downcasted to T^+, simply forgetting we know they will never be null, hence $T^- \preceq T^+$. Therefore, any expression of type T^- can be assigned to a variable of type T^+, but to go the other way around requires a test.

Problems arise when taking into account the initialisation of fields. Suppose a class C declares a field f with a non-null type T^-. If a variable c is declared of type C^-, we expect $c.f$ to denote a non-null value. This would be easy to ensure if we knew that c was fully initialised, but during the construction of an object, the object being constructed is accessible using the `this` keyword, and the field f may not be initialised yet, hence `this.f` points to null. In fact, until it has been assigned a non-null value, the language semantics requires that `this.f` denotes the null pointer: when an object is allocated, all fields are set to null. Worse, `this` can be used as an argument for another method, and then $x.f$ (x being the corresponding method parameter x of type C^-) may denote a null pointer whereas f has been declared T^-. Moreover, all these method can be redefined by descendants of the class C, thus even if the methods of a class are implemented such that no uninitialised field is dereferenced, redefining methods could have unexpected results.

To solve this problem, Fähndrich and Leino introduce a new type C^{raw-} denoting non-null but partially initialised value, illustrated in Listing 2.3. To make sure that once a field of an object of type C^- is initialised it never contains null again, they require that expressions assigned to $c.f$ be of type T^- (hence non-null) even if c is of type C^{raw-} (hence not fully initialised). This strong requirement of *monotonicity* ensures the soundness of the type system even in the presence of aliases or in a concurrent setting. Finally, by the end of every constructor of a class C, every field annotated with a non-

```
class C {
   field T⁻ f; // f is supposed not null
   C () {
   // at this point, this is typed as C^raw−
   f := new Object (); // f is initiated to non null
   // at this point, this is typed as C⁻
   }
}
```

Listing 2.3: Example of a constructor annotated with raw types

null type and directly declared in class C, must have been assigned. This ensure that for any class C, `new` C can safely be downcaste from C^{raw-} to C^{-}.

Inference. Annotations are useful tools to specify invariants when they are known, but annotating all the information needed by a verification tool on a real program is a daunting task. It can be alleviated by using *default annotations policies*, but even a default policy is not sufficient to find correct types for all programs. Static type inference allows to avoid manual annotation. Hubert *et al.* have proposed an automatic null pointer analysis for inferring non-null annotations of fields [HJP08] in the type systems proposed by Fähndrich and Leino [FL03]. In order to track down the initialisation state of fields, they are using an instrumented semantics which annotates field with the status *def* (defined) as soon as their are initialised. Furthermore, they proved the correctness of the proposed analysis with respect to the operational semantics of the core object-oriented language on which the analysis is formalised, providing a semantic foundation of non-null annotations, and that proof has been machine checked using the Coq [BC04] proof assistant. They also proved the completeness of the analysis in the sense that on any typable programs the analysis will prove the absence of null dereferencing without any hand-written annotation.

2.5.3 Conclusion

In the present chapter, we introduced the theory of *abstract interpretation* and how a static analyser can be formalised in this theory. We instantiated this formalism on two family of analyses: numerical analyses and analyses of the heap. To provide a formal definition of programs to analyse we also introduced two core languages and their semantics. The result certification of numerical analyses will be studied in Chapter 5, and the approach will be adapted to analyses of the heap in Chapter 6.

Chapter 3

Deductive verification background

The previous chapter presented the static analysis context in which our approach is grounded. But our methodology is based on Verification Condition generation, a concept central to deductive verification. Therefore before the presentation of our approach in the chapters 4 to 6, we recall in the present chapter the necessary background to understand our approach and to evaluate its originality.

We first recall in Section 3.1 the seminal works of Floyd, Hoare and Dijkstra. Then Section 3.2 discusses some challenges brought forth by modern programming languages and described the solutions implemented in state-of-the-art tools. Finally, Section 3.3 discusses the key concept of *trusted computing base* and presents one answer to the problem of trust relevant to our problem: *proof-carrying code*.

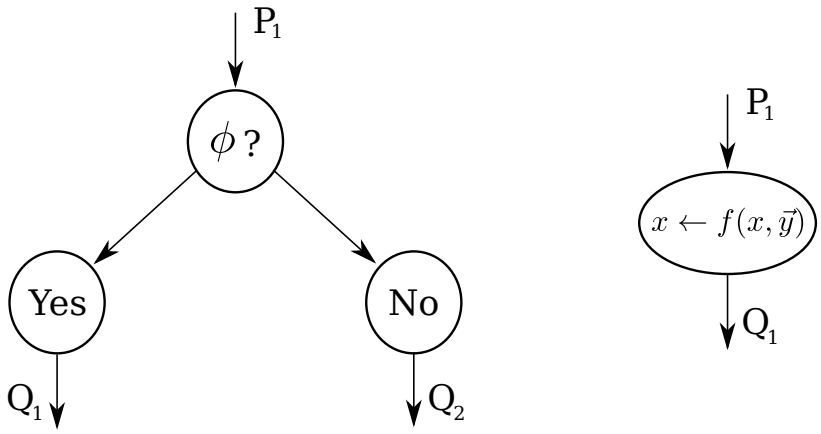
3.1 Deductive verification

3.1.1 Axiomatic semantics

The first efforts towards a formal definition of the meaning of programs, and the first attempts to prove the absence of errors in a program, led in the late 60's to several, very different, paradigms of program semantics. In particular, the work of Robert W. Floyd was among the first to use proof techniques as a formal definition of programming languages. In his seminal work on *Assigning Meanings to Programs* [Flo67], he relates the semantics of a language to the tools used to verify properties of programs:

> A semantic definition of a particular set of command types, then, is a rule for constructing, for any command c of one of these types, a verification condition $V_c(P; Q)$ on the antecedents [pre-conditions P] and consequents [post-conditions Q] of c.

In other words, the semantics of a program—represented as a flowchart—is a set of verification conditions: formulae parametrised by annotations on each edge of the flowchart. If, when the pre-condition—*i.e.*, the condition on entrance—of a statement is true, the post-condition—*i.e.*, the condition on exit—is true after execution of the statement, we say the annotations are correct. And by construction the annotations are correct only if the verification conditions are valid formulae.



a) Test with a condition ϕ

b) Assignment of a variable x, with a term f, $\vec{y}$ being a list of variable distinct of x

Figure 3.1: Flowchart representation of a test and an assignment, annotated with pre-conditions P_i and post-conditions Q_i

For example, the verification condition $VC_c(P_1; Q_1, Q_2)$ for the test depicted in Figure 3.1 is

$$(P_1 \wedge \phi \vdash Q_1) \wedge (P_1 \wedge \neg\phi \vdash Q_2)$$

where the semantics of the relation $\vdash$ is given by a particular deductive system, which includes the axioms and rules of inference of the first-order logic with equality. This verification condition corresponds to the usual semantics of a test: if P_1 is true before the test on ϕ, then for the annotations to be verified, Q_1 must be a logical consequence of $P_1 \wedge \phi$ and Q_2 must be a logical consequence of $P_1 \wedge \neg\phi$. This *forward* reasoning—assume the pre-condition, deduce a constraint on the post-condition—leads to a semantics of the assignment introducing existentially quantified variables. Using the notation of Figure 3.1, the semantics of $x \leftarrow f(x, \vec{y})$ is, if P_1 has the form $R(x, \vec{y})$

$$(\exists x_{old}, x = f(x_{old}, \vec{y}) \wedge R(x_{old}, \vec{y})) \vdash Q_1$$

The existentially quantified x_{old} represents the variable x before the assignment. Assuming $R(x, \vec{y})$ holds before the statement, we know that $R(x_{old}, \vec{y})$ still holds after the assignment to x, and we also know that the new value of

x is $f(x_{old}, \vec{y})$. For the annotations to be verified, we must prove that these facts are sufficient to deduce Q_1. Note that this semantics is only consistent if we suppose the language free of aliases, and while this is fairly easy to prove for purely functional expressions, it requires further precautions in the presence of side-effects. Section 3.2 gives more details on the way modern verification frameworks alleviate this problem, and Section 4.3.2 explains how that solution is adapted in our approach.

This definition of the semantics is a formal basis for proofs of relations between input and output, and can be extended to account for the termination of a program, using *variants*—*i.e.*, equations on terms belonging to well-ordered sets. Floyd also remarks that all verification conditions have the same form

$$VC_c(P; Q) \equiv T_c(P) \vdash Q$$

where $T_c(P)$ is a formula depending on the statement c and on the precondition P, and whatever Q is, for the annotations to hold it must be the case that $\vdash T_c(P) \implies Q$. Therefore $T_c(P)$ is the *strongest verifiable consequent*, and as it can be calculated using the semantics and the precondition, not all edges need to be annotated for the program to be verified, only entrance points and one edge per loop. This combined with mechanical theorem proving techniques opened the way for automatised verification of programs.

Expanding on Floyd's work, Tony Hoare introduce a specific notation—now called *Hoare triples*—to link textual representation of programs and logical specification of the states of execution. The triple $\{P\}Q\{R\}$[1] is interpreted as

> If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion.[2]

Rather than having rules to produce verification conditions depending on the statements and proving the verification condition in a deductive system, Hoare expands the deductive systems with rules and axioms dealing with triples [Hoa69], providing a full-fledged *axiomatic semantics* of a programming language usually referred to as the *Hoare-Floyd logic*.

For example, proving the triple

$$\{P\}x \leftarrow f(x, \vec{y})\{Q\}$$

would require the use of the axiom for the assignment and of the rule for consequences. First, an axiom for assignment describe the semantics of the

[1] The initial notation by Hoare read $P\{Q\}R$ but we use instead the pervasive $\{P\}Q\{R\}$.

[2] This sentence refers to *partial correctness*, that is, P does not implies that Q terminates. *Total correctness* requires that if P is true then Q terminates and R is true on its completion.

statement

$$\frac{}{\vdash \{Q[f/x]\}x \leftarrow f\{Q\}}$$

then an inference rule links the notion of consequence in the logical world to the program annotations

$$\frac{\vdash P \implies Q[f/x] \quad \vdash \{Q[f/x]\}x \leftarrow f\{Q\}}{\vdash \{P\}x \leftarrow f\{Q\}}$$

The semantics of the assignment is here given in a *backward* manner, whereas Floyd used a *forward* presentation, but the intent is the same, and in both cases the semantics is only consistent for a side-effect free language.

Remark that one of the premises of the consequence rule is a purely logical judgement, and is left to be proved using the standard first order logic deductive system. Indeed, the pre-condition P is a *sufficient* condition but not a *necessary* one. The *weakest* condition that must be true on the state of execution, before the assignment, for the post-condition Q to hold is $Q[f/x]$.

3.1.2 Weakest precondition calculus

This notion of *weakest pre-condition* has been systematised by Edsger W. Dijkstra and he uses it to reformulate and tighten the Hoare-Floyd logic as a *predicate transformer semantics* [Dij75].

Considered as a function of the post-condition, the weakest pre-condition, written $\mathrm{wp}(C, Q)$ for a program C and a post-condition Q, is an example of *predicate transformer* and gives the semantic of the program Q. To quote Dijkstra [Dij97, Chap. 4, p. 24]

> While the semantics of a specific mechanism (program) are given by its predicate transformer, we consider the semantics characterisation of a programming language given by the set of rules that associate the corresponding predicate transformer with each program written in that language. From that point of view we can regard the program as "a code" for a predicate transformer.

The rules of inference introduced by Hoare are replaced by rules associating a program to its predicate transformer, given by a syntax driven *weakest pre-condition calculus*

$$\begin{aligned} \mathrm{wp}(\text{“}x \leftarrow f\text{”}, Q) &= Q[f/x] \\ \mathrm{wp}(C_1; C_2, Q) &= \mathrm{wp}(C_1, \mathrm{wp}(C_2, Q)) \\ &\dots \end{aligned}$$

and proving that a triple $P\{C\}Q$ holds amount to prove in the first order logic deductive system that

$$\vdash P \implies \mathrm{wp}(C, Q)$$

A dual *strongest post-condition* calculus can be defined, and generalises Floyd's remark on the strongest verifiable consequent.

3.2 Deductive verification of modern languages

More recent work has built upon Dijkstra's calculus to provide more efficiency [FS01], but the core of most modern *verification condition generators* (VCgen) is still a weakest pre-condition calculus. Nonetheless, extending Hoare-Floyd logic to modern programming languages give rise to all kinds of problem, *e.g.*, aliases between variables, dynamic dispatch during method call.

3.2.1 Memory model

Aliases between variables occur when two variables refer to the same memory unit, thus, when assigning one variable, the value stored in memory changes for both variables. A semantic of aliases can be given by the definition of a *memory model*, *i.e.*, a refined abstraction of the link between a variable and the memory manipulated by the program, that allows an explicit semantic definition of assignment that takes aliases into account.

In object-oriented languages, aliasing occurs within the heap where all objects are stored: each object is a collection of fields which may contain pointer to objects in the heap, thus when two variables o and o' contain objects, a modification to $o.f$—f being a field of o—also modifies $o'.f$ if o and o' reference the same object.

Rather than modelling the memory by a map from variable names to values, a common practice, initiated by Cartwright and Oppen [CO81], is to define an abstraction of the memory , the domain of *memory locations*[3], and to maintain two separate maps, as illustrated on Figure 3.2: one map from locations to values (the memory) and one from variables to locations (the store). Assigning a value v to a variable x is defined as mapping x to a location l and assigning v to l. Hence if x is an alias of another variable y, then both x and y refer to the same location, and assigning a new value to either variable, effectively change the value behind both.

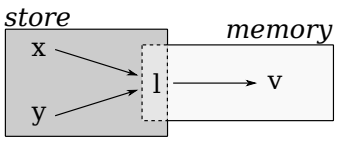


Figure 3.2: Simple memory model

[3]The term employed in [CO81] was *abstract addresses* but we reformulate in term of *locations* to be consistent throughout the dissertation.

In type-safe object-oriented languages—*e.g.*, JAVA, C#—aliasing occurs only when two pointers of same type point to the same object, therefore proving that the types are different ensures absence of alias. A simple memory model represents the heap by a mapping from locations to objects, objects being mapping from field names to value. Most verification frameworks use a variation upon this approach depending on the language, the reasoning engine and design choices.

In unsafe languages such as C, objects or structs can overlap in memory, memory can be explicitly deallocated, and pointers can be forged using arithmetic operations—under certain restrictions. However, most program in C that need to be verified adhere to strict coding rules and type discipline, as after all type-safe programs can be written in unsafe languages. Following this principle, the VCC [CDH+09] verification environment provide a memory model for concurrent C [CMTS09] that allows easy typed reasoning and more cumbersome but feasible untyped reasoning.

3.2.2 Framing

Even if a memory model allows precise and sound reasoning about aliases, reasoning about procedure calls is usually based on contracts: the state before and after a call are related using the pre-condition and post-condition of the procedure, not its implementation. If the local environment is supposed untouched by the call—apart from a given variable storing the result—the heap is not restored after the call: any modification of the heap done during the call remains valid after the call. The area of the heap that a procedure modifies is called the *footprint* of the procedure, and a conservative but sound approximation of the footprint is to consider the whole heap altered by the call. However, a more precise over-approximation of the footprint may be needed for verification.

The most common approach, called *framing* [BMR95], is to use *framing conditions* to specify, for each procedure, which part of the heap it is allowed to change during its execution. The main challenge is to define a logic expressive enough to state conditions precise enough for the specifications of "most" programs to be provable. But another challenge emphasised by numerous verification approaches is to make the logic "usable", which entails keeping annotations concise, and making conditions simple enough to be specified by non-specialist programmers.

The JAVA Modelling Language [LBR06] (JML) is a specification language used by numerous verification environment targeting JAVA, *e.g.*, KRAKATOA [MPMU04, MPm05], ESC/JAVA [FLL+02], KEY [ABB+00]. Each method should define, apart from pre-condition (`require`) and post-condition (`ensure`), its frame (`assignable`), using JAVA expressions. A common solution to enhance the expressivity of such expressions is to allow the definition of *ghost variables and instructions*, *i.e.*, variables used solely in an-

notations, modified only by ghost instructions and never used to alter the control-flow or the data of the program. Ghost variables are the basic mechanism in more complex framing schemes, such as the *ownership* system used in the VCC [CDH+09] and Spec# [BLS04] verification environments, or the *dynamic frames* [Kas06] used in the Dafny [Lei10] environment.

In the *separation logic* approach, used for example in the VeriFast [JSP10, JSP+11] verification environment for C and Java, rather than introducing a special framing condition to the specification of methods, the logic of annotations is extended with a new operator $*$ that allows logical formulae to hold for distinct parts of the memory. The axiomatic semantics of the language is changed to use this operator and a new deductive system is introduced. In this approach, the framing condition is, in a sense, inferred from the pre-condition.

3.2.3 Intermediate Verification Languages

Both the memory model and the framing conditions complicate substantially the definition of the VC generator (VCgen). In particular, a precise memory model leads to axioms that must be taken into account while proving the VCs and may require the program to be modified so as to reference explicitly the memory model constructs, and the interpretation of framing conditions leads to new verification conditions, may require additional analysis of the program and introduce some axioms. To separate the WP calculus and the definition of the memory model, numerous verification environment rely on *Intermediate Verification Languages* (IVL), *e.g.*, Why [BFMP11] and BoogiePL [BCD+05].

The IVL provides a simple programming language in which the source code must be encoded and specification language in which the annotations—including the framing conditions and explicit side-effects—and the memory model must be encoded. Such languages are not meant to be executable, and parts of the program and of the memory model may be axiomatised. In place of a compiler or an interpreter, the language comes with a WP calculus and a back-end in charge of sending the VCs to automated theorem provers and interpreting the results. A deductive verification environment for a specific language can be seen as the combination of the IVL, its WP calculus and a compiler of the initial language to the IVL.

Figure 3.3 presents an overview of the deductive verification scheme based on intermediate verification languages. Squared nodes stand for the different stages of the work-flow of the verification process, and rounded nodes stands for input/output of the different processes used. Nodes are annotated by relevant existing tools: the VCC [CDH+09] verification environment takes a C program translates it into BoogiePL, so does Spec# [BLS04] for C# and Dafny [Lei10] for its specific language, and Krakatoa [MF09] translates a Java program into Why.

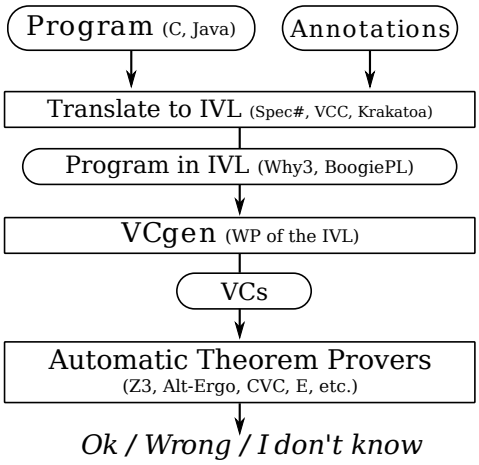


Figure 3.3: Overview of deductive verification approaches based on IVLs

The LOOP compiler [vdBJ01] is based on a different but comparable approach. It translates JAVA programs to theories in the logic of the ISABELLE/HOL or PVS Proof-Assistants (PA), adds user assertions about classes and prove them against the theories. The memory model is described using construct of the logic and JAVA block statement are translated according to their denotational semantics into state transformers [HJ00]. In this approach, a specific logic is used as an intermediate language between JAVA and the logic of the PA, verification conditions are generated as theorems in the PA and are proved using tactics and decision procedures implemented in the PA rather than using automated theorem proving.

3.2.4 Automated deductive verification

To provide a usable framework for program verification, the generated verification conditions must be proved automatically with good reliability, and a large part of recent successes in program verification is due to progress in the *Automated Theorem Proving* practice. We summarise the different steps in the verification of a program to justify this claim.

1. The program to be verified has to be annotated with pre-conditions, post-conditions, framing conditions, loop invariants and variants. The annotation process can be tedious and requires knowledge of both the verification process—and its mathematical foundation—and the program itself—or the algorithm behind it. For verification to be practical and accessible to programmers with no particular theoretical expertise, the specification language has to include high-level constructs the programmer is familiar with, and a default policy must be defined such that most annotations can be left implicit.

2. The annotated program is translated into an Intermediate Verification Language. Again, for practical reasons this translation must be automatic and require the minimum amount of additional information. At this point the annotations have to be made explicit and translated in the specification language used by the weakest pre-condition calculus. These translations may generate numerous axioms to account for the high-level constructs of both the input programming language, the initial specification language and the memory model of the source language.

3. Verification conditions are generated automatically by a weakest pre-condition calculus. The preceding exposition concentrated on verification condition corresponding to the Hoare triple

   $$\{\text{pre-condition}\}\text{program}\{\text{post-condition}\}$$

   but they also account for array bound checking, absence of arithmetic overflows, various data-structure invariants, absence of null pointer dereferencing, validity of framing conditions, etc. Therefore, the WP calculus typically generate a large number of formulae. They may be, at the same time, rather "easy" to prove—in term of proof length for example, or of the complexity of the reasoning involved—and large in term of number of symbols.

4. These verification conditions have to be discharged, *i.e.*, proved *valid*. Apart from toy examples, a pen-and-paper proof is unrealistic: there are too many formulae involved, with too many symbols, and the process would be far too error prone. The use of an interactive proof-assistant can alleviate some problems. Formulae can be manipulated with ease, partial automation simplifies the process, and the consistency of individual proof steps can be checked. However, it requires a high degree of expertise from the user and the proof effort grows—figuratively at least—exponentially with the size and complexity of the program. To avoid any manual interaction in the proof search, the verification condition may be discharged by *Automatic Theorem Provers* (ATPs), in which case another translation may be needed—from the verification conditions' logic to the input logic of the ATP.

At all stages, automation is key to make program verification practical. However, at the last stage the ATP may fail, and a negative response or the absence of response may happen for different reasons. The prover may need an unreasonable amount of time or resources to finish the proof, or the translation to the ATP's logic may not be complete. In any case, the verification process may be stuck at this stage even if the program and the annotations *are* correct. To finish the proof, the user can try to change the

annotations, in hope that a different equivalent/weaker/stronger formula may be discharged more easily; he can try to add more annotations, effectively cutting the problem into simpler ones; he can try to use a different ATP, as different algorithms can succeed on different formulae in practice; or he can do the remaining proofs "by hand", if there are not too many left.

3.3 Establishing trust

As we saw, automated program verification is a complex process relying on a number of pieces of software implementing different theoretically sound but non trivial transformations and decisions. An error at any stage, whether due to a human error in the specification of to a buggy implementation, can have a devastating effect on the final result if it remains undetected.

3.3.1 Trusted Computing Base

Trusted Computing Base (TCB) is a term coined by John M. Rushby and a key concept to assess the security guaranties of computer systems [Rus81]. It can be defined, in the terms a famous free collaborative encyclopedia puts it, as

> The set of all hardware, firmware, and/or software components that are critical to the security of the system, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardise the security properties of the entire system.[4]

Securing the TCB of a system typically involves manual or computer-assisted software audit, extensive testing, and/or program verification—in its broader meaning—using formal methods. But automated program verification—by means of verification condition generation and automated theorem proving as detailed in Section 3.1—involves a number of large piece of software and non trivial decisions. An error during any stage of the verification process may result in a false assessment that the verified program meets is specification, thus the TCB of a verification framework should be carefully identified and examined to determine i) which decisions and process are key to the soundness of the final result, and ii) how to limit and facilitate the verification of that result.

The first step of verification is the annotation of the program. It may involve the manual definition in a formal specification language of the specification of the program and of the program invariants, or the translation of the manual annotations from one specification language to another. If an error is done at this stage, and the program's post-condition is not strong enough to ensure the informal specification of the program, the verification

[4] Wikipedia, Trusted computing base, March 18th, 2013.

process may return a faulty assessment. However, if the error result in a wrong loop annotation, which is not kept invariant during the execution of the program, the weakest pre-condition calculus will output an non-provable verification condition, and the verification process will correctly fail. Thus the annotated pre-condition and post-condition of the procedures/methods are part of the TCB, but not the loop invariant. For the verification of any property, it is impossible to annotate automatically all programs, and there will always be a human interaction at this stage or some form of incompleteness of the process. To limit the risk of error during the manual specification process, standard security policies can be used—policies specified once and for all and applied to all programs, thus included in the TCB.

The second step is the translation of the program in an equivalent alias-free form, using a memory model. If the translation is not correct, the verification can obviously not be trusted. Moreover, a deductive verification framework relies on an automatic translation, and even if the translation's formalisation is correct it may be implemented incorrectly, therefore the implementation have to be included in the TCB. For similar reason, so is the memory model and the automatic verification generator used in the third step.

In the final step, the verification are discharged. As explained earlier, this will involve some automatic process, whether it is an interactive or a fully automatic proof search. If an interactive proof search is used, then the resulting proof must be checked. If the search is fully automatic, the prover is part of the TCB.

At all stage, if an automatic translation or proof is required, the TCB may contain, in place of the translation/proof search software, a *result verifier*. While discharging the verification conditions for example, the ATP can be treated as an *untrusted software*: it is not part of the TCB, but a separated software—the so-called verifier—will be used to check that the proof found is consistent—*w-r-t* the appropriate deductive system—and is a proof of the validity of the verification condition. The verifier is therefore included in the TCB, but is usually easier to prove correct than the ATP.

In any case, neither sound theoretical foundations nor formal proof of correctness of an individual step are sufficient to ensure the reliability of a program verification framework. That does not mean that resistance is futile, nor that formal verification is a lost cause. It is mainly a reminder that software complexity can only be partially tamed and remain wild at heart, and an opportunity for continuous research advance.

3.3.2 Proof-Carrying Code

The use of proof-verifiers to eliminate the proof-search from the TCB is one of the key aspect of George C. Necula's Proof-Carrying Code (PCC) concept [Nec97], whose initial problem is closely related to our own and

whose solution may be adapted to our needs. The starting point of PCC is its seminal application scenario.

> A *code consumer* must somehow become convinced that the code supplied by an untrusted *code producer* has some (previously agreed upon) set of properties. Sometimes this is referred to as establishing "trust" between the consumer and producer. [Nec97]

To solve this problem, the PCC approach consists in splitting the verification process into two stages. First, the code producer creates a *safety proof* that the code respects a safety policy formally defined and made publicly available by the code consumer. Then the code consumer uses a proof verifier to make sure this proof is valid.

Checking the consistency of the proof is not enough: the code consumer has to check that it is a proof of the appropriate property—the safety of the code at hand. This means that if the program verification is based on verification conditions, both the consumer and the producer have to perform the verification condition generation. The producer uses it to produce a proof, and the consumer uses it to check that the carried proof attests to the fact that the code respects the safety policy. For example, the carried proof can be a set of deduction trees using the rules of a—previously agreed upon—deductive system, in which case the proof verifier is composed of a verification condition generator and a type checking kernel that checks i) that each node of a tree is a consistent application of the deductive system and ii) that each verification condition is the root of a proof tree.

The TCB on the code consumer side includes the safety policy and the proof verifier, but neither the code nor the proof. Therefore, the proof verifier has to be trustworthy, which means in most cases that the underlying decision procedure should be simple and that its implementation should be small. The program in charge of producing the verifications conditions, the *VCgen*, can be shared by, or different for, the producer and the consumer, but in any case, must be included in the TCB of the consumer.

For practical reasons, the proof verifier should also be fast, at least fast enough that it can be done on the device executing the code in a reasonable time. This requirement seems attainable when considering the intuition that

Proof-checking is easier than proof-search.

but it may still be challenging when applying PCC to code distributed on portable device with limited resources when compared to a standard workstation. Moreover, the previous intuition may be misleading when proof-objects are very large and when the prover does not need to maintain it during the search, or when the proof-checking is executed in a system whose safety has a high computing cost. Conversely, eliminating the proof-search from the TCB grants more leeway to use aggressive optimisations and unsafe heuristics, effectively expanding the reach of the ATPs.

PCC applications. The seminal work by Necula and Lee [NL98c] attached the annotations used to generate VCs and the proofs of the VCs—obtained using the Touchstone theorem prover [NL00] and encoded in the Edinburgh Logical Framework [HHP93, NL98b]—to programs, compiled from a type-safe subset of C to optimised DEC Alpha machine code. This approach requires the use of a certifying compiler [NL98a], able to generate annotated machine code from annotated programs. Necula *et al.* later on applied this methodology to a subset of Java and developed an optimising certifying compiler targeting Intel x86 architectures [CLN$^+$00], showing that PCC can be applied to object-oriented languages with high-level features such as exceptions and floating-point arithmetic.

Morissett *et al.* developed a related approach to machine code safety that relied on *Typed Assembly Language* [MWCG99] (TAL). In this approach, annotations are restricted to source-level type information and certifying compilers generate TAL code. The strongly-typed assembly code can then be verified by a type checker, and processed by the standard MASM assembler to generate Intel x86 machine code. In comparison with the PCC approach developed by Necula, type-checking replaces VC generation and requires no additional proof, thus the machine code is only packed with types rather than both annotations and proofs, but the extent of property that can be ensured is limited to type information whereas PCC can use higher-order logic to define arbitrary invariants.

3.3.3 Foundational Proof-Carrying Code

The Trusted Computing Base of the TAL and PCC approaches include sophisticated type-systems and VCgen. Their soundness can be proved w-r-t the semantics of the languages, and the seminal works provide extensive pen-and-paper proofs of the underlying rules, but obtaining a formal proof of their implementation remains a challenge.

The Foundational Proof-Carrying Code approach [App01] (FPCC) has been proposed by Appel and Felty [AF00] to alleviate this concern. The operational semantics of machine instructions and the safety policy are described in a higher-order logic, and the proof that a program abide by the safety policy must be given in this logic. There is no Verification Condition generation, as any typing or Hoare-Floyd rule must be proved in the logic as a lemma to be used.

> The proof must explicitly define, down to the foundations of mathematics, all required concepts and explicitly prove any needed properties of these concepts. [App01]

Therefore, the TCB of a FPCC approach includes only the verifier of the higher-order logic, the semantics of the machine instructions and the safety policy.

Appel and its co-authors have described successive semantic models [AF00, AM01, AAV02] to prove the soundness of increasingly expressive type systems. All definitions and proofs were encoded in the Edinburgh Logical Framework, implemented and machine-checked in the TWELF meta-logic framework [PS99]. A major difficulty of these FPCC approaches is the difficulty to obtain a soundness proof. To alleviate this problem, Hamid *et al.* [HST+03] proposed to use type derivations and syntactic soundness proofs [WF94]—rather than semantic soundness proofs—and used COQ to develop and prove such a FPCC framework. A last example of an FPCC approach in this line of thoughts is the TALT project [Cra03], that revisit previous TAL approaches[5] and expand them with a more expressive type theory, an operational semantics closer to actual hardware and a fully machine-checkable proof defined in TWELF.

Another possible approach to FPCC is to provide machine-checkable proofs for the VCgen of a PCC framework. The VERYPCC project [WNKN04, WN05] conducted Nipkow *et al.* aims to develop a PCC framework in the ISABELLE/HOL theorem prover, where VCs are generated by a certified generic VCgen as theorems in the proof-assistant, and proofs are ISABELLE proof scripts to be replayed. The framework was used to develop PCC environments for a simplified assembly language and for the JAVA Virtual Machine (JVM). The Mobile Resource Guarantees framework [AGH+05] (MGR) is another PCC environment for the JVM, but dedicated to proofs of absence of run-time resource violation. It is based on a powerful resource-aware type system proved in ISABELLE/HOL, and requires programs to be attached proofs—again, ISABELLE proof scripts—of bound preservation.

The Open Verifier [NS03, CCNS05] is another project aiming at sound VCgens to implement FPCC architectures for proving memory safety. However, it focuses on providing a general framework allowing the definition of customised verifiers and type-systems without compromising the TCB of the approach. A new type checker is an untrusted module which, using a scripting language, instructs the trusted kernel of a proof strategy for discharging the verification conditions.

Finally, Vogels *et al.* [VJP10] recently presented a fully formal, machine-checked proof of the soundness of a realistic VCgen, implemented in COQ. This project was initially part of an effort to improve the reliability of deductive verification approaches based on IVLs, but as a clear value for a FPCC approach. Herms *et al.* [HMM12] achieved a similar result for the VCgen of an IVL inspired by the WHY platform. There implementation in COQ is structured such that it can be extracted into a standalone executable VCgen that can target different ATPs.

[5] TALT stand for TAL Two.

Chapter 4

Static analysis result certification methodology

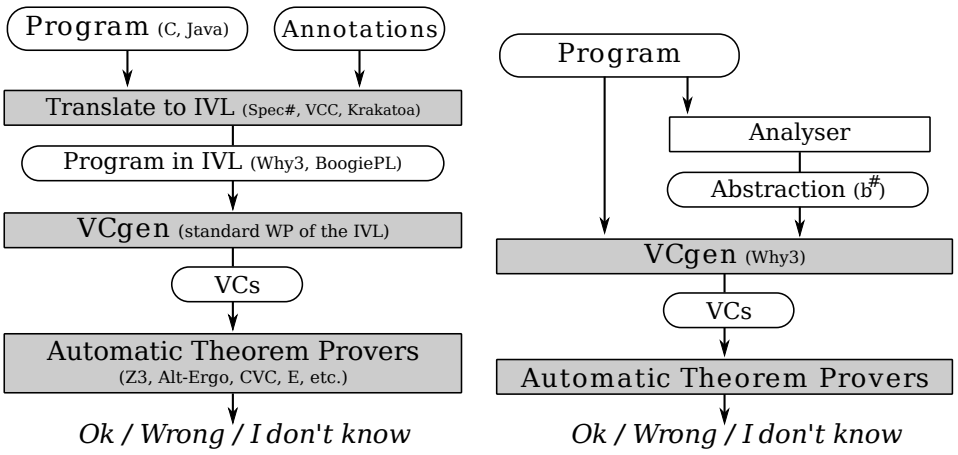
The present chapter details our approach and how we apply the deductive verification concepts and tools presented in Chapter 3 to the certification of the results of the analyses presented in Chapter 2. First, Section 4.1 presents an overview of the approach, and discusses its possible applications. Section 4.2 details the Many-Sorted First-Order Logic as the logical framework in which is based the generation of VCs, and presents the specification of a memory model in this framework. Then Section 4.3 presents the specifics of the approach: how to use the formalisation of an analyser in the abstract interpretation framework to translate the abstraction of a program into annotations of that program and alleviate the framing problem, what VC can be used to certify the result of an analyser, and the general methodology we use to translate an operational semantics into a VC calculus. Finally, Section 4.4 applies this methodology to the core languages presented in Section 2.2 and Section 2.4.

4.1 Approach

The idea of using static analyses in a verification environment is certainly not new. Most of the verification frameworks presented in Chapter 3 use abstract interpretation, and static analysis in general, to infer parts of the specifications of programs and to reduce the amount of manual annotations needed to verify a program. However, the problem addressed in this dissertation is not how to use static analysis to help deductive verification but how to use deductive verification to check the results of a static analyser, thus our approach side-steps some difficulties of deductive verification approach, such as the framing problem.

4.1.1 Overview of the approach

The starting point of the approach comes from the observation that the abstraction $b^\sharp$ of a program is interpreted using the concretisation function $\gamma : \mathcal{R}each^\sharp \to \mathcal{P}(\mathcal{S}tate)$ as the description of the possible execution states before each program point, thus corresponds to annotations of the program *w-r-t* to the property the analyser is trying to establish. Furthermore, the fact that $b^\sharp$ is a correct post-fixpoint means that the information at each program point is sufficient to ensure the pre-condition of the following program point, and $b^\sharp$ proves the absence of runtime error only if the information at each program point ensures that no runtime error may occur. These properties of the abstraction can be checked using a Verification Condition calculus, where $b^\sharp$ is considered as an oracle providing untrusted annotations of the program.



a) Overview of deductive verification **b)** Overview of result certification

Figure 4.1: Comparaison between the classical deductive verification scheme and the result certification scheme

In this context, to build a result certification scheme we can use standard deductive verification techniques, taking the safety property the analyser was checking as the only specification of the program. Figure 4.1a recalls an overview of the deductive verification process , and Figure 4.1b presents how it can be modified in an overview of our approach by replacing annotations of the program by the result of an analyser. The similarities with the deductive verification process goes as far as the Trusted Computing Base (TCB) of the approach (figured by grey boxes on the figures), which in both cases includes the Automated Theorem Prover and the VCgen, but not the annotations of the program: the only annotations we trust are those specifying the semantics of "absence of runtime errors". However, the TCB of our approach contain no translation of the program into the

IVL: we rely on a dedicated Verification Condition calculus—described in Section 4.3—specified and proved sound *w-r-t* the semantics in the IVL, as detailed in Chapter 5, therefore the program is represented *explicitly* and not translated, and the abstraction $b^\sharp$ is *not* just translated into an existing annotation language, *e.g.*, JML for JAVA.

Remark that we do not aim at proving that a program meets its functional specification, only that the verdict of an analyser on a program is correct. The properties proved are not specific to a program, and the precision of the "annotations" is bounded by what the abstract domain of the analyser can represent. Moreover even if the static analysis problem in general is undecidable, we deal with the results of the analyser, *i.e.*, we are interested in programs where the analyser succeeded to decide a property, at least partially. Therefore we expect the result verification to be fully automatic. Even if the abstraction calculated on a program is not precise enough to prove the complete absence of error, we should be able to prove that it is a correct over-approximation of the program's behaviours. Nonetheless, our approach depends on Automated Theorem Provers for the final verdict. Even if these tools are mature and powerful, the satisfiability problem remains at least NP-hard[1]: they can run out of time or memory, and may not be able to discharge the VCs even if they are valid, hence the possible *I do not know* verdict.

Our research problem and the means we use to address it are closely related to the PCC approach. Following Necula, we choose Floyd style verification condition generation as the core mechanism of our result certification scheme. However, contrary to Floyd, we want to specify the semantics of the analysed language as an operational semantics, against which the verification conditions soundness should be established. Contrary to seminal PCC approaches [Nec97], the only proof carried by a program is the abstraction calculated by the analyser, and contrary to the Foundational Proof Carrying Code approach [App01]—which takes an extreme stands regarding what can be included in the TCB—we attack the problem under a more practical angle, and contend ourselves with ATPs in the TCB. The problem of trust in the result of ATPs will be examined in Chapter 8.

To help relate the VC calculus with the operational semantics, we build a memory model in the *many sorted first-order logic* framework (see Section 4.2). It allows us to manipulate memory states in a functional way (as maps) in the exact same way the operational semantics does. Furthermore, defining memory states as a sort belonging to the logic and that can be manipulated as a data structure gives us a direct way to generate the VCs from the operational semantics described as an interpreter, which eliminates from the TCB any form of translation from the input languages to an Intermediary Verification Language (see Chapter 5) and establishes the soundness of

[1] Or worse, depending on the theories added to propositional logic.

the VC calculus *w-r-t* to the operational semantics.

4.1.2 Applications

Our approach to the result certification of static analyses may have several applications, and each application may have its own constraints on the size of the TCB and the efficiency of the tools.

PCC approach to an existing uncertified static analyser. Either the code producer or the user of the analyser—which may or may not be the same entity—wants to provide a checker. In this case, the result certification is performed by the user of the analyser. We suppose the certification of the analyser itself—*i.e.*, its proof—is not possible: either the user does not possess sufficient expertise, or the producer—which may possess sufficient expertise—wants to provide a verifier at a low cost. In any case, both producer and user agree on the specification language of the results of the analyser and on the semantics of the language. Once the verifier is implemented, result checking should be fully automatic. The constraints on the TCB may be high or low depending on the requirements of the code user, as may be the efficiency requirements. A specific application of the seminal PCC approach is the problem of trust in third-party software on portable devices, in which case the user's main requirement comes from the limited resources available on the device. If the limited resources prevent the user from executing ATPs, the scheme must be expanded to include a proof certification component: the producer provides proofs for the VCs in addition to the abstraction of the program, and the user runs the VCgen on the program and the abstraction, and a proof verifier on the VCs and the proofs provided by the producer. Chapter 8 presents an approach to the certification of SMT proofs.

Certification of critical software application. Static analyser's results are used to prove safety properties during the certification process of critical software, therefore the analyser itself must be held to the highest degree of scrutiny. However, it is a complex program, and code review or standard certification processes may be as costly as avoiding the use of an analyser altogether. Time is not an issue, and the user of the verifier may be an expert in program verification, but safety policies are non negotiable and the verification should meet the highest certification requirements. It entails much scrutiny of the TCB of the verifier, and probably disqualify ATPs from being part of it. Chapter 8 provides the beginning of a solution to replace ATPs by a verifier programmed in the Coq [BC04] proof-assistant.

Testing during static analyser implementation. The developer of the analyser wants to check the analyser's results soundness and precision

during the development phase of the analyser. The verifier is used as an oracle for testing the analyser. The verifier is also used while prototyping the analyser to quickly check the precision of the analysis. Once the input are specified, the checking should be fully automatic and fast, to allow back-and-forth iterations between modifying the analyser and testing the consequences of these modification. The verifier should be easily modifiable and the soundness of these modifications should be easy to check against the operational semantics. The analyser's results should also be easy to enter. This application requires responsive tools, which means that the first concern is efficiency. A trade-off between a completely safe but very slow and a barely correct but very fast verifier must be reached, thus including ATPs in the TCB is not a problem, and using different ATPs concurrently is probably desirable.

4.2 Specification language

To detail the translation from abstract states to predicates, and present the VC calculus, we need to specify the logic for stating and proving the verification conditions.

4.2.1 Many-sorted FOL

In the Hoare-Floyd logic presented in Section 3.1.1, the core logic was First-Order Logic (FOL) with equality, augmented with specific syntax and rules for Hoare triples, and the VC generated by the weakest pre-condition belong to FOL with equality, and arithmetic for the core numerical language. If we want to use SMT solvers to discharge the VC, we must either state the VC in their logic, or define a translation to it, but as we want to prove the soundness of the VC calculus *w-r-t* an operational semantics, we want the logic used for the VC to be as close as possible to the semantics defined in Chapter 2.

Fortunately, the whole point of Satisfiability *Modulo Theory* is to provide powerful decision procedures for formulae written in *rich* logic, namely *many-sorted FOL*. *Rich* is deliberately non-committing, as many-sorted FOL is not necessarily more expressive: in the absence of interpreted symbols (*e.g.*, arithmetic), any many-sorted formula can be linearly encoded in FOL and *vice versa* FOL is trivially *mono*-sorted. However, the richer syntax is not reducible to syntactic sugar either, as it allows the use of efficient decision procedures—more details on these are given in Chapter 8. Again, efficient does not mean the underlying problem belongs to a different complexity class, but refers to efficiency *in practice*.

Sorts are distinct sets of symbols and variables, they are more commonly referred to as types in programming languages and we will use both terms

indistinctly. Many-sorted formulae are built upon well-typed terms, and valuations must respects types as well. formal definitions and the applications can be found in a survey by Meinker and Tucker [MT93] or in the definition of the SMT-LIB 2.0 input standard for SMT solvers [STB10]. In the following, we use the convention of the SMT-LIB standard, the interpretation of FOL it uses, and illustrate the logic using examples.

In this formalism, new sorts can be defined, and the behaviour of objects belonging to those sorts can be given using axioms, effectively defining *theories*. Following this principle we describe the memory models of our languages as theories of semantic states based on finite sorts describing the syntactic domains, *i.e.*, $\mathcal{PP}$, $\mathcal{Var}$, $\mathcal{Var}_{\mathcal{A}}$, $\mathcal{Method}$ for the core numerical language and $\mathcal{PP}$, $\mathcal{Var}$, $\mathcal{Method}$, $\mathcal{Class}$ and $\mathcal{F}$ for the core object-oriented language. These are finite sorts, instantiated using the syntax of the program to verify, and can be easily defined in FOL, but to describe the higher-order constructs used in the semantic domains, such as environments $\mathcal{Env} = \mathcal{Var} \to \mathcal{Val}$ or heaps $\mathcal{Heap} = \mathcal{L} \to \mathcal{Val}$, we need to present the pervasive theories of equality and *Uninterpreted Functions* (UF[2]), and *maps*.

The UF theory formalises the minimum amount of information on functions needed to make deduction on them. It states that equality is an equivalence relation—reflexive, symmetric and transitive—and that function application should at least satisfy the axiom of congruence. These axiom can be given directly in FOL.

$$\forall x, x = x \qquad \forall x, y, y = x \implies x = y \qquad \forall x, y, z, x = y \wedge y = z \implies x = z$$

$$\forall x, y, x = y \implies f(x) = f(y)$$

Remark that we omitted the sorts. The many-sorted FOL does not suppose the existence of polymorphic operators and functions[3], and one axiom per sort and per function symbol should be given. However, the UF theory is one of the core theory of many-sorted logic and of SMT solvers—if only for equality— thus the symbol of equality is supposed shared among all theories and the UF axioms include a *built-in* quantification of function symbols.

The theory of maps can be viewed as an extension of UF. This theory introduce two new function symbols, *get* and *set*—noted $\bullet[\bullet]$ and $\bullet[\bullet \leftarrow \bullet]$. This theory is sometimes called *array* theory as it defines the *functional constructs* without side-effects that can be used together with a memory model to give the semantics of arrays. It was introduced—to our knowledge—in the seminal work on using memory models to alleviate the restriction on aliasing in the Hoare-Floyd logic [CO81]. The version given here correspond to the formalisation used in most modern work. The axioms state what

[2]Sometimes referred to as EUF.

[3]Note that the logic of the WHY3 tool suite proposes an encoding of polymorphism in many-sorted FOL [BP11].

happen when doing a selection (get) on a previously assigned map (set)

$$\forall m : \text{map } \mathcal{V}ar\ \mathcal{V}al, (a_1, a_2) : \mathcal{V}ar^2, b : \mathcal{V}al, a_1 = a_2 \Rightarrow m[a_1 \leftarrow b][a_2] = b$$
$$\forall m : \text{map } \mathcal{V}ar\ \mathcal{V}al, (a_1, a_2) : \mathcal{V}ar^2, b : \mathcal{V}al, a_1 \neq a_2 \Rightarrow m[a_1 \leftarrow b][a_2] = m[a_2]$$

We gave the axioms with the sorts $\mathcal{V}ar$ and $\mathcal{V}al$, to define the sort "map $\mathcal{V}ar$ $\mathcal{V}al$" of maps from variables to values, but these formulae are more accurately described as axiom patterns or schemes to be instantiated when necessary. Using these two axioms, relations between maps can be modelled, and using a new symbol *const* and a third axiom

$$\forall b : \mathcal{V}al, a : \mathcal{V}ar, const(b)[a] = b$$

new maps can be created.

4.2.2 Theory of semantic states

Using these two theories and the sorts describing the syntax, we can define the sorts of environments, heaps, and eventually states, as presented in Figure 4.2a for the numerical language and in Figure 4.2b for the object-oriented language. Following the notations of the semantics, we use records—which can be viewed as tuples with a special notation for access functions—to represent the Cartesian product, but will build records as tuples, *i.e.*, with implicit field names.

$$\begin{aligned}
\mathcal{V}al &= \mathbb{Z} \\
\mathcal{E}nv &= map\ \mathcal{V}ar\ \mathcal{V}al \\
\mathcal{A}rray &= \{len : \mathbb{N}\ ;\ elt : map\ \mathbb{N}\ \mathcal{V}al\} \\
\mathcal{E}nv_{\mathcal{A}} &= map\ \mathcal{V}ar_{\mathcal{A}}\ \mathcal{A}rray \\
\mathcal{S}tate &= \{st : \mathcal{E}nv\ ;\ ars : \mathcal{E}nv_{\mathcal{A}}\ ;\ cpp : \mathcal{PP}\}
\end{aligned}$$

a) Theory of numerical semantic states

$$\begin{aligned}
\mathcal{V}al &= \mathcal{L} \cup \{null\} \\
\mathcal{E}nv &= map\ \mathcal{V}ar\ \mathcal{V}al \\
\mathcal{O}bj &= \{cl : \mathcal{C}lass\ ;\ obj : map\ \mathcal{F}\ \mathcal{V}al\} \\
\mathcal{H}eap &= map_{\perp}\ \mathcal{L}\ \mathcal{O}bj_{\perp} \\
\mathcal{S}tate &= \{st : \mathcal{E}nv\ ;\ hp : \mathcal{H}eap\ ;\ cpp : \mathcal{PP}\} \\
\mathcal{E}rr &= \{NullPointer, LookupFail\}
\end{aligned}$$

b) Theory of object-oriented semantic states

Figure 4.2: Theories of semantic states

The sort $map_{\perp}$ corresponds to a specialisation of maps where the constant map can only be defined with the value $\perp$. Otherwise, a fully allocated infinite memory could be defined, which would jeopardise the coherence of the system: some VCs suppose that a fresh address can always be found, therefore if it is not possible, that is, if a fully addressed heap can be defined, the VC is always *true*. If a VC is always *true*, it can be used to prove anything, notably *false*.

Once these theories are defined, formulae containing variables belonging to semantic domains can be given a semantics. For example, in the pre-

viously defined theories, the following (artificial) closed formula ϕ holds—denoted using standard notation by $\models \phi$.

$$\begin{aligned}\phi \equiv \quad & \forall h : \mathcal{H}eap, \forall e : \mathcal{E}nv, \forall X : \mathcal{V}ar, \forall l_1, l_2 : \mathcal{L}^2, \forall v_1, v_2 : \mathcal{V}al, \\ & e[X] = l_1 \wedge e[X] = l_2 \wedge h[l_1 \leftarrow v_1][l_2] = v_2 \implies v_1 = v_2\end{aligned}$$

Indeed, using the axioms of equality, we can deduce from $e[X] = l_1 \wedge e[X] = l_2$ that $l_1 = l_2$, hence the locations l_1 and l_2 are equal. Then using map axioms we can prove that $v_1 = v_2$: v_2 is the value in the heap $h[l_1 \leftarrow v_1]$ at location l_2 and as $h[l_1 \leftarrow v_1]$ is the heap h where the value v_1 has been assigned to location l_1, the first axiom of maps allows us to conclude.

Once all syntactic domains are defined, algebraic types[4] can be used to define the sorts of expressions, tests and statements, using one constructor per operator and per statement. We also suppose that the logic includes *interpreted functions*[5] that we will used to define the encoding of flowchart program in the exact same way they were defined in Sections 2.2 and 2.4, using functions such as *get_stmt* and so on. We call assertions, noted $\mathcal{A}ssn$, the set of formulae built upon semantic and syntactic domains as sorts.

This theory of semantic states is straightforward and indeed implied when stating the semantics, but once formally stated and backed up by decision procedures, it allows—contrary to Hoare-Floyd logic—explicit manipulation of states in assertion. The standard remark that a formula with free variables denotes a set—the set of all valuations that evaluate the formula to *true*—leads in the cases of states to an easy characterisation of set of states. If we note

$$\mathcal{A}ssn(x_1 : \mathrm{sort}_1, \ldots, x_n : \mathrm{sort}_n)$$

the set of formulae with n free variables $x_1, \ldots, x_n$ of sorts $\mathrm{sort}_1, \ldots, \mathrm{sort}_n$, then the set $\mathcal{A}ssn(s : \mathcal{S}tate)$ denotes a subset of $\mathbb{P}(\mathcal{S}tate)$, *i.e.*, all states that can be represented by formulae. We will write $\mathcal{A}ssn(s)$ when there is no ambiguity on the sort of s, and the fact that a formula $\phi \in \mathcal{A}ssn(s)$ evaluates to *true* on a state s_0—*i.e.*, on the valuation that maps the logical variable s to the value s_0—will be written $s_0 \models \phi$.

4.3 Methodology for the result certification of a static analyser

We are interested in verifying an analysis result, *i.e.,* we consider given an untrusted $b^\sharp$—recall that $b^\sharp$ is an abstraction describing all program points—that we shall validate by discharging verification conditions. To do so, we

[4]See [Pas09] for the translation of algebraic data types in many-sorted FOL.

[5]Depending on the tool used to decide the many-sorted FOL, different constructs may be defined by default. If functions are not accessible, they can be represented by maps or axiomatised.

first translate the abstract states into assertions, to be used as pre-condition and post-condition to produce VCs.

4.3.1 Compiling abstract states into assertions

Once the core logic—in which verification conditions are stated and proven—has been specified, we need, to obtain Floyd-style VCs, to transform an element $b^\sharp$ into pre-conditions and post-conditions expressed through functions $pre, post : \mathcal{PP} \to \mathcal{A}ssn(s)$ satisfying

$$\begin{aligned} s &\models pre(p) \text{ iff } s \in \gamma(b^\sharp) \wedge s.cpp = p \\ s &\models post(p) \text{ iff } s \in \gamma(b^\sharp) \wedge s.cpp \in succ(p) \end{aligned}$$

To get an effective way of computing pre and $post$, we require elements of Abs to be mappings from program points to assertions ($Abs = \mathcal{PP} \to \mathcal{A}ssn(s)$) and define $\gamma(b^\sharp) = \{(e, h, p) \mid (e, h) \models b^\sharp(p)\}$. For example, the formalisation presented in Section 2.3.1 of an interval analyses can be readily expressed this way. An interval would be defined as an assertion in $\mathcal{A}ssn(v : \mathcal{V}al)$. The elements of $\mathbb{D}^{int}$—the interval abstract domain, see Section 2.3.1—presented as pairs, $\bot$, or $\top$, can be translated to assertions by the function π_{val} defined as

$$\begin{aligned} \pi_{val} &: \mathbb{D}^{int} \to \mathcal{A}ssn(v : \mathcal{V}al) \\ \pi_{val}([a, b]) &= a \leq v \wedge v \leq b \\ \pi_{val}(\bot^{int}_{\mathbb{D}}) &= false \\ \pi_{val}(\top^{int}_{\mathbb{D}}) &= true \end{aligned}$$

Note that the codomain of π_{val} is the set $\mathcal{A}ssn(v : \mathcal{V}al)$ of all formulae with a free variable v.

The abstraction of reachable states $\mathcal{R}each^\sharp$ would be defined as a function from program points to assertions in $\mathcal{A}ssn(s : \mathcal{S}tate)$. The elements of $\mathcal{R}each^\sharp_{int}$ can be translated to functions in $\mathcal{PP} \to \mathcal{A}ssn(s : \mathcal{S}tate)$ by the function $\pi_{\mathcal{R}each}$ defined as follows. It takes as argument an abstraction $b^\sharp_{int}$: a function that returns an interval from a program point p and a variable X.

$$\begin{aligned} \pi_{\mathcal{R}each} &: \mathcal{R}each^\sharp_{int} \to \mathcal{PP} \to \mathcal{A}ssn(s : \mathcal{S}tate) \\ \pi_{\mathcal{R}each}(b^\sharp_{int}) &= p \mapsto \bigwedge_{X \in \mathcal{V}ar} \pi_{val}(b^\sharp_{int}(p, X))[s.env(X)/v] \end{aligned}$$

For all program point p, the assertion $\pi_{\mathcal{R}each}(b^\sharp_{int})(p)$ is a finite conjunction over $\mathcal{V}ar$ where the Xs are constants, hence s is the only free variable besides the parameter p. Each atom of the conjunction is the translation of an interval into an assumption: the function π_{val} returns an assumption with a free variable v that is substituted for the term with a free variable s. For example, if the variable `x` at the program point p is annotated with an interval

$[a, b]$, then $\pi_{val}([a, b]) = a \leq v \leq b$ and the final atom is $a \leq s.env(\mathtt{x}) \leq b$. A similar translation can be defined for the polyhedral analysis, but as a polyhedron can already be represented by a conjunction of linear inequalities, we will not go into further details.

More generally, given a sufficiently powerful assertion language $\mathcal{A}ssn$ (*e.g.,* many-sorted FOL), most concretisations can readily be expressed this way. However, the verification conditions for even the simplest object-oriented program analyses are neither quantifier-free, nor do they fall into existing decidable fragments. Our experiments show that current SMT solvers are usually incapable of discharging such verification conditions. Nevertheless, this approach is sufficient for numerical analyses as established in Chapter 5, and will be used as a foundation for the soundness of a refined approach for object-oriented analyses presented in Chapter 6.

4.3.2 VCgen methodology

With fully annotated programs represented as flowcharts, we are now ready to present a simple verification condition generation algorithm (VCgen). The informal semantics of the VCgen is that, for a given annotated program, all the verification conditions should be valid if and only if any execution starting in a state verifying the annotated pre-condition never reaches an error state. As accessibility is defined using a small-step semantics, the reachable states are also defined for non-terminating executions, therefore we ensure the absence of errors even if a program does not terminate.

As the assertion language is powerful enough to represent sets of states, and as we can manipulate—build and relate—explicitly states, the premises and conclusions of the inference rules of the small-step semantics can effectively be translated into assertions of $\mathcal{A}ssn(s : \mathcal{S}tate)$. However, terms of the form $(s.env, e) \Rightarrow v$ and $init \rightarrow^* end$—parts of the semantics that are not strictly small-step—are not part of the assertion language and have to be translated. The formers—terms involving the natural semantics of expressions—are easy to transform into formulae $[\![e]\!](s)$ while computing the VCs, the latter—terms involving big-step reductions—will be abstracted by the pre-conditions and post-conditions of methods.

We define for each semantic rule $\mathtt{rule_i}$ a function $VC_{\mathtt{rule_i}}$ which takes as arguments a program point and the closed terms of the corresponding statement, and returns an assertion in $\mathcal{A}ssn(s)$. We will write $terms(\mathtt{s})$ the closed terms of the statement $\mathtt{s}$ to simplify notations, *e.g.*, if $\mathtt{rule_i}$ can be applied at point p, we will write the VC it generates as follows.

$$VC_{\mathtt{rule_i}}(p, terms(\mathtt{get_stmt(p)}))$$

For instance, if a program p_0 is annotated by a statement $X := Y$, the rule to be applied is `Assign` and the generated VC would be $VC_{\mathtt{Assign}}(p, X, Y)$. The pre-conditions and post-conditions come from assertions in $\mathcal{A}ssn(s)$ denoted

by $pre(p)$ and $post(p)$. The post-condition usually should be checked on an updated state s', thus the corresponding assertion is $post(p)[s'/s]$—the assertion $post(p)$ where s' is substituted at free occurrences of s.

To signify when both $post(p)$ and $pre(p)$ denote a sub-assertions—and therefore can be reduced, *i.e.*, partially or completely evaluated—we put them between $[\![\ ]\!]$. Remark that we supposed that the logic includes *interpreted functions*, therefore, as the sort $\mathcal{PP}$ is finite, we can encode the functions *pre* and *post* directly in the logic and the reduction denoted by $[\![\ ]\!]$ is not mandatory. However, it simplifies the formulae substantially, and it seems relevant to point out where such reduction can take place.

Informally, the guiding idea behind the VCs is to establish for each program point p that the following holds

$$\forall s, s' : \mathcal{State}, [\![pre(p)]\!] \Rightarrow s \rightarrow s' \Rightarrow [\![post(p)[s'/s]]\!]$$

To translate $s \rightarrow s'$ to the assumption language, we state the relation between s and s' in the logic, supposing that the conditions of the corresponding semantic rule hold. If several rules may apply for a particular program point—depending on the statement annotating the node—we produce one VC per semantic rule. The results of $VC_{\texttt{rule}_\texttt{i}}$ functions are assumptions in $\mathcal{Assn}(s)$, leaving only the variable s free: all other variables are bound. We could boldly omit all universal quantification at the head of formulae, as to prove the validity of a formula $\forall x, \phi$ we ask ATPs to prove the unsatisfiability of $\neg\phi$, however to clarify the formulae—*e.g.*, make explicit sorts and binding point—we will only treat differently the variable s. When necessary, the closed formed of a VC—including the universal quantification over s—will be noted $\overline{VC}_{\texttt{Rule}_\texttt{i}}(\ldots)$.

Finally, a specific VC quantify over all program points, and check that, at each program point, a state satisfying the post-condition will satisfy also the pre-condition of the next statement.

$$\begin{aligned} VC_{continue} = \forall p \in \mathcal{PP}, s \in \mathcal{State}, \\ p \neq p^+ \wedge post(p) \wedge s.cpp \in \texttt{succ}(p) \Rightarrow pre(s.cpp) \end{aligned}$$

The atom $p \neq p^+$ specifies that p is not an exit point. Note that p is not directly used in the precondition: it is assumed that the current program point is a successor of p. It does not matter that the implication $post(p) \Rightarrow pre(s.cpp)$ may not be true for other states s: the semantics ensures that in all executions, after the execution of a the statement annotating a program p, the current program point is a successor of p.

To define the soundness of a VC calculus such that it can be embedded in the standard definition of a result certification approach, we write $\mathcal{VC}_{gen}(P, b^\sharp)$ the set of verification conditions generated on a program $P \in \mathcal{Pgm}$ encoded as a flowchart, given a analysis result $b^\sharp$ encoded as *pre* and *post* predicates.

Definition 4.3.1. A VC calculus is defined as a function returning a set of VC from a program and an abstraction.

$$\mathcal{VC}_{gen} : \mathcal{R}each^\sharp \times \mathcal{P}gm \to \mathcal{P}(\mathcal{A}ssn)$$

Definition 4.3.2 (Soundness of a VC calculus)**.**
A VC calculus $\mathcal{VC}_{gen}$ is said *sound* if and only if

for all well-formed program P and untrusted analysis result $b^\sharp$, if all the verification conditions are valid

$$\forall \Phi \in \mathcal{VC}_{gen}(P, b^\sharp), \Phi \text{ is valid}$$

the absence of run-time error is guaranteed by $b^\sharp$.

$$\forall s \in \mathcal{S}tate, e \in \mathcal{E}rr, s \in \mathcal{R}each(P) \Rightarrow s \not\rightsquigarrow e$$

4.3.3 Handling semantic invariants

Proposition 2.4.1 states that the semantics preserves the well-formedness condition (predicate *wf*) and that consecutive states are compatible (predicate $compat(s, s')$). As an analyser may rely on this semantic invariant for its soundness, it may be necessary to discharge the VCs, *i.e.*, the annotations may not be sound without the semantics invariants.

The proposition holds for states that are initially well-formed, therefore well-formedness must be in the hypothesis of the VCs. It can be systematically stated along the hypothesis of all semantic rules, or added to closed VCs:

$$\overline{VC}_{\texttt{rule}_\texttt{i}}(\ldots) = \forall s \in \mathcal{S}tate, wf(s) \implies VC(\ldots)$$

For most VCs, the state after a statement is fully describe *w-r-t* the state before, therefore the invariant holds by construction as a consequence of the axioms of the theory of semantic states. However for VC^{post}_{call}, the big-step reduction of the SOS is abstracted by the pre-condition and post-condition of the procedure/method called, therefore the invariant does not hold by construction of the two states *init* and *end* but by induction on the derivation of semantic states. In this case, the compatibility and well-formedness of both states should be part of the relation abstracting the big-step reduction, and needs to be stated in the hypotheses of the VC.

$$VC^{post}_{call}(\ldots) = \begin{cases} \forall \quad init, end \in \mathcal{S}tate^2, \\ \quad [\![pre(p)]\!] \Rightarrow \ldots \\ \quad\quad compat(init, end) \wedge wf(init) \wedge wf(end) \Rightarrow \ldots \\ \quad\quad\quad \Rightarrow [\![post(p)[s'/s]]\!] \end{cases}$$

This skeleton of VC dedicated to verification of the post-condition of call program points makes explicit the compatibility between the initial call state and the end call state. We leave the details of the specification of *init* and *end* to Sections 4.4.1 and 4.4.2, where the general scheme of the VCs are instantiated on the numerical and object-oriented core languages.

4.4 VCgen for core languages

This section applies the methodology previously detailed to both core languages used in the experiment. As explained in Section 4.3.2, these VCs are closely related to the operational semantics of the languages described in Section 2.2 and Section 2.4, but to avoid bloating this section with inference rules, most of the semantic rules are not reproduced.

Following the notation of Section 4.3.2, we put between brackets $[\![\ ]\!]$ the sub-assertions that can be reduced when computing the VCs. Terms of the syntax of the languages are denoted by capital letters, to distinguish them from logical variables.

4.4.1 Numerical VCgen

Expressions. The translation of the natural semantics of expressions into assertions, presented in Figure 4.3, needs for the numerical core language two different functions: one for numerical expressions and one for Boolean expressions. They are recursive but trivially terminating, and as they have the same purpose we will use the same notation: $[\![\bullet]\!](\bullet)$.

$$[\![e]\!](s) = \begin{cases} n & \text{if } e = n & n \in \mathbb{Z} \\ s.env(\mathtt{X}) & \text{if } e = X & X \in \mathcal{V}ar \\ [\![e_1]\!](s) \oplus [\![e_2]\!](s) & \text{if } e = e_1 \oplus e_2 & \oplus \in \{+, -, \times\} \end{cases}$$

$$[\![t]\!](s) = \begin{cases} \mathit{true} & \text{if } t = \mathtt{true} \\ \mathit{false} & \text{if } t = \mathtt{false} \\ \neg[\![t_1]\!](s) & \text{if } t = \mathtt{not}\ t_1 \\ [\![t_1]\!](s) \wedge [\![t_2]\!](s) & \text{if } t = t_1\ \mathtt{and}\ t_2 \\ [\![e_1]\!](s) \bowtie [\![e_2]\!](s) & \text{if } t = e_1 \bowtie e_2 & \bowtie \in \{=, \neq, <, \leq\} \end{cases}$$

Figure 4.3: Translation of the natural semantics of expressions

This function does not compute the result of an expression, it translate a term of the syntax into a term of the logic. Thus it takes as arguments a numerical expression (respectively a Boolean expression) and a *variable name* of sort $\mathcal{S}tate$, and returns a term of sort $\mathcal{V}al$ (respectively an assertion) that leaves free the *variable name* denoting a state, whereas the natural semantics $(e, s) \Rightarrow v$ is to be understood as a relation between an expression (respectively a test) e, a state s and a value in $\mathbb{Z}$ (respectively $\{\mathtt{true}, \mathtt{false}\}$).

Assignments and tests. Figure 4.4 details the VC rules for assignments. It is a direct translation of the SOS, as all conditions of the rule are part of the assertion language. The function $VC_{\mathtt{Assign}}$ producing the VC for an assignment takes as argument three terms of the syntax: the program point

$$VC_{\texttt{Assign}}(P, X, E) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ let\ p' = s.cpp^{+}\ in \\ let\ env' = s.env[X \leftarrow [\![E]\!](s)]\ in \\ let\ s' = (env', s.ars, p')\ in \\ \quad [\![post(P)[s'/s]]\!] \end{cases}$$

Figure 4.4: VC function for assignments

P which is annotated by the statement, a variable name X and an expression E coming from the statement $X := E$.

$$\text{Assign}\frac{\texttt{get_stmt}(s.cpp) = X := E \qquad (s.st, E) \Rightarrow n}{s \longrightarrow (s.st[X \leftarrow n], s.ars, s.cpp^{+})}$$

The semantic rule for the assignment (recalled here for readability) establishes the link between the state before execution of the statement and after: the new environment is an update of the previous environment at variable X with the natural semantics of the expression E, and the current program point is incremented. Accordingly, the VC is valid only if the environment of state s' is an update of the environment of state s at variable X with a value that is a model of the term $[\![E]\!](s)$, and the current program point in state s' is the successor of the program point in s.

$$VC^{\top}_{\texttt{JumpIf}}(P, T, P_j) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ [\![T]\!](s) \Rightarrow \\ let\ p' = P_j\ in \\ let\ s' = (s.env, s.ars, p')\ in \\ \quad [\![post(P)[s'/s]]\!] \end{cases}$$

$$VC^{\bot}_{\texttt{JumpIf}}(P, T, P_j) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ \neg[\![T]\!](s) \Rightarrow \\ let\ p' = s.cpp^{+}\ in \\ let\ s' = (s.env, s.ars, p')\ in \\ \quad [\![post(P)[s'/s]]\!] \end{cases}$$

Figure 4.5: VC functions for tests

Remark that we consider the *let* ... *in* construct as part of the logic, and use it to simplify formulae. We do not generate VCs corresponding to syntactic constraints (such as X *is assignable*, *i.e.*, X is not one of the read-only variables, $\texttt{p}_0$ and $\texttt{p}_1$), as they can be checked while building the VCs. If X is not assignable (syntactically) then the VC generation fails (or asks for a proof of *false*).

There are two semantic rules for tests, therefore we generate two VCs per node annotated with a `JumpIf` T P_j statement, using the VC functions

described in Figure 4.5. The operational semantics states that the execution should jump to program P_j if the test T evaluates to true. Accordingly, the VCs for a `JumpIf` statement states that the post-condition should hold on a state s' with a current program point equal to P_j if the assertion $[\![T]\!](s)$ is valid, and with a current program point equal to $s.cpp^+$ if the assertion $\neg[\![T]\!](s)$ is valid.

Procedure calls. The semantics of procedure calls uses a big-step reduction to avoid using a call stack. The corresponding VC functions, presented in Figure 4.6 on the next page, abstract this big-step reduction by the contract of the procedure, *i.e.*, its pre-condition and post-condition. Recall that a procedure call takes the form

$$X := F(E_0, E_1)$$

It takes two arguments, two numerical expressions, thus the call-related VC functions has five terms of the program syntax as arguments: the program point P, the variable name X that stores the result of the call, the procedure name F, and the two arguments of the call.

An initial state *init* is built using the arguments of the call. The initial environment env_{init} for numerical variables of the *init* state is built

$$s.env[p_0 \leftarrow [\![E_0]\!](s)][p_1 \leftarrow [\![E_1]\!](s)]$$

and its initial program point is defined as the entry point $F_{\emptyset}$ of the procedure. Remark that $F_{\emptyset}$ is a shorthand for the term returning the correct starting point given a procedure name. We could have used a function name to make it explicit but we preferred to keep the notation as compact as possible in the SOS and in the VCs. The same goes for F_{∞}. However, as the numerical core language does not have any dynamic dispatch feature, these terms can be evaluated while calculating the VCs, hence the brackets $[\![\ ]\!]$.

A state *end* is supposed compatible with *init*—*i.e.*, the formal parameters have not been reassigned—and should validate the post-condition. The *compat* predicate is defined according to the operational semantics. Recall that no *well-formed* property is defined for the numerical core language, hence no predicate *wf* is necessary. The state *end* is used as if it were the final state of execution of the procedure,

$$\begin{array}{l} \ldots \\ end.cpp = [\![F_{end}]\!] \Rightarrow \ldots \\ let\ env' = s.env[X \leftarrow end.env[res]]\ in\ \ldots \end{array}$$

i.e., its current program point is supposed to be the exit point F_{end} of the procedure, and it is used to defined the environment env' of the exit state.

According to the operational semantics, there is only one way to define the initial state of a procedure call given the environment at call point,

$$VC^{pre}_{\texttt{Call}}(P, X, F, E_0, E_1) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ let\ env_{init} = s.env[p_0 \leftarrow [\![E_0]\!](s)][p_1 \leftarrow [\![E_1]\!](s)]\ in \\ let\ init = (env_{init}, s.ars, [\![F_0]\!])\ in \\ \quad [\![pre(F_0)[init/s]]\!] \end{cases}$$

$$VC^{post}_{\texttt{Call}}(P, X, F, E_0, E_1) = \begin{cases} \forall end : \mathcal{State}, \\ [\![pre(P)]\!] \Rightarrow \\ let\ env_{init} = s.env[p_0 \leftarrow [\![E_0]\!](s)][p_1 \leftarrow [\![E_1]\!](s)]\ in \\ let\ init = (env_{init}, s.ars, [\![F_0]\!])\ in \\ \quad [\![pre(F_0)[init/s]]\!] \Rightarrow \\ \quad [\![post(F_\infty)[end/s]]\!] \Rightarrow \\ \quad end.cpp = [\![F_\infty]\!] \Rightarrow \\ \quad compat(init, end) \Rightarrow \\ \quad let\ p' = s.cpp^+\ in \\ \quad let\ env' = s.env[X \leftarrow end.env[res]]\ in \\ \quad let\ s' = (env', s.ars, p')\ in \\ \quad\quad [\![post(P)[s'/s]]\!] \end{cases}$$

Figure 4.6: VC functions for procedure calls

$$VC^{no\ error}_{\texttt{GetArray}}(P, X, Y, E) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ \quad 0 \leq [\![E]\!](s) < s.ars[Y].len \end{cases}$$

$$VC^{no\ error}_{\texttt{SetArray}}(P, X, E_1, E_2) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ \quad 0 \leq [\![E_1]\!](s) < s.ars[X].len \end{cases}$$

$$VC_{\texttt{GetArray}}(P, X, Y, E) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ let\ p' = s.cpp^+\ in \\ let\ v = s.ars[Y].elts[[\![E]\!](s)]\ in \\ let\ env' = s.env[X \leftarrow v]\ in \\ let\ s' = (env', s.ars, p')\ in \\ \quad [\![post(P)[s'/s]]\!] \end{cases}$$

$$VC_{\texttt{SetArray}}(P, X, E_1, E_2) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ let\ p' = s.cpp^+\ in \\ let\ l = s.ars[X].len\ in \\ let\ elt = s.ars[X].elts\ in \\ let\ elts' = elts[[\![E_1]\!](s) \leftarrow [\![E_2]\!](s)]\ in \\ let\ ars' = s.ars[X \leftarrow (l, elts')]\ in \\ let\ s' = (s.env, ars', p')\ in \\ \quad [\![post(P)[s'/s]]\!] \end{cases}$$

$$VC_{\texttt{Length}}(P, X, Y) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ let\ p' = s.cpp^+\ in \\ let\ env' = s.env[X \leftarrow s.ars[Y].len]\ in \\ let\ s' = (env', s.ars, p')\ in \\ \quad [\![post(P)[s'/s]]\!] \end{cases}$$

Figure 4.7: VC functions for array statements

therefore the state *init* is defined using the *let in* notation. However, the state *end* can be any state satisfying the post-condition of the procedure and compatible with *init*, and depending on the post-condition, there could be many such states, thus the $VC^{post}_{\texttt{Call}}$ quantify universally over *end*.

Arrays. The VC functions for statements dealing with arrays, presented in Figure 4.7 on the preceding page illustrate how VC can be added to deal with the *blocking* semantics: the conditions $VC^{no\ error}_{\texttt{GetArray}}$—for accesses in arrays—and $VC^{no\ error}_{\texttt{SetArray}}$—for updates in arrays—will only be valid if the pre-condition ensures that no *array out-of-bound* error may occur.

Definition 4.4.1. We write $\mathcal{VC}_{num}$ the VC calculus defined by the rules presented in Figures 4.4, 4.5, 4.6 and 4.7:

Let P be a program encoded as a flowchart and $b^\sharp$ be the result of an analyser encoded as *pre* and *post* predicates.

- If P is not well-formed, then $false \in \mathcal{VC}_{num}(P, b^\sharp)$
- $\forall p \in \mathcal{PP}$, if $rule_i$ may apply on $\texttt{get_stmt}(p)$, then

$$\overline{VC}_{\texttt{rule}^{\texttt{k}}_{\texttt{i}}}(p, terms(\texttt{get_stmt(p)})) \in \mathcal{VC}_{num}(P, b^\sharp)$$

- $VC_{continue} \in \mathcal{VC}_{num}(P, b^\sharp)$

Theorem 4.4.1 (Soundness)**.**
The verification condition calculus $\mathcal{VC}_{num}$ is sound:

Let P be a well-formed program and $b^\sharp$ be the untrusted result of an analysis. If all the verification conditions are valid

$$\forall \Phi \in \mathcal{VC}_{num}(P, b^\sharp), \Phi \text{ is valid}$$

the absence of run-time error is guaranteed by $b^\sharp$.

$$\forall s \in \mathcal{State}, s \in \mathcal{Reach}(P) \Rightarrow s \not\rightsquigarrow OutOfBound$$

Proof. Theorem 4.4.1 has been proved using the intermediate verification language Why3:

- the VC calculus is specified as a set of lemmas parametrised by the implementation of a flowchart,
- the operational semantics of the language is specified as an interpreter,
- and the lemmas are used to discharge—with ATPs in this case—the proof obligations generated by the weakest pre-condition calculus of Why3 for the interpreter.

More details on the implementation can be found in Section 5.2. □

4.4.2 Object-oriented VCgen

The VC functions for assignments and tests are similar to the ones used in the numerical case and will not be detailed.

Expressions. As the expressions are very simple in the core object-oriented language, the translation of the natural semantics of expressions into assertions only need one (non-recursive) function and is more a shorthand than anything else.

$$[\![E]\!](s) = \begin{cases} null & \text{if } E = \texttt{null} \\ s.env(\texttt{X}) & \text{if } E = \texttt{X} \end{cases}$$

Heap accesses and updates. Figure 4.8 on the next page presents the VC functions for the semantic rule of accesses to the heap. One for the normal case, $VC_{\texttt{Getfield}}$ and one for the null pointer error case, $VC^{NullP}_{\texttt{Getfield}}$. In case the analysis cannot prove the absence of null pointer dereferencing, this latter VC will not be validated. These VCs are not quantified only over states, because we can not use the *let in* to name the value of a variable—it is either $\perp$ or a location and we need to distinguish these two cases—or an object in the heap—the fact that all accessible locations point to a non-null object is a semantic invariant, not a syntactic property.

A heap access takes the form $X := Y.F$, where X and Y are variable names and F is a field name, hence the four arguments of the VC function—the program point P is the first argument as in previous VC functions.

The VC functions for the semantic rules of updates in the heap presented in Figure 4.9 on the following page are similar. The order of the arguments of the VC functions reflects the syntax of a heap update statement $X.F := Y$.

Method calls. A method call takes the form $X := Y.C_0.M(A_0, A_1)$, where X is the variable storing the result, Y if the variable pointing to the object on which the call is made, C_0 is the first class—the highest in the class hierarchy—defining the method M, and will be used as the highest class in the hierarchy when searching for an implementation, M is the method name, and A_0 and A_1 are the arguments of the method—either `null` or a variable given the limited expressions of the language.

Figure 4.10 on the next page presents the relation $s \rhd_{c_0,m} (init, p_{end})$ used in the call-related semantic rules. It is translated to a conjunction of constraints linking a state s and the initial state *init* of a method. We write $[\![s \rhd_{C_0,M} (init, p_{end})]\!](Y, A_0, A_1)$ this translation. The translation contains an axiomatising of the lookup algorithm: the class c_{look} returned by the lookup is a super-class of the dynamic class c in which the method is implemented, but a subclass of the class C_0 of the call.

The dynamic class is defined by the literal $s.hp[l] = (c, o)$, where l is defined as the location of the object calling the method using the literal

$$
\begin{array}{rcl}
VC_{\texttt{Getfield}}(P,X,Y,F) & = & \left\{\begin{array}{l}
\forall o \in \mathcal{Obj}, l \in \mathcal{L}, c \in \mathcal{Class}, \\
\quad [\![pre(P)]\!] \Rightarrow \\
\qquad s.env[Y] = Loc(l) \Rightarrow \\
\qquad s.hp[l] = (c,o) \Rightarrow \\
\qquad\quad let\ v = o[F]\ in \\
\qquad\quad let\ p' = s.cpp^{+}\ in \\
\qquad\quad let\ env' = s.env[X \leftarrow v]\ in \\
\qquad\quad let\ s' = (env', s.hp, p')\ in \\
\qquad\qquad [\![post(P)[s'/s]]\!]
\end{array}\right. \\
VC^{NullP}_{\texttt{Getfield}}(P,X,Y,F) & = & \left\{\begin{array}{l}
\forall l \in \mathcal{L}, \\
\qquad [\![pre(P)]\!] \Rightarrow \\
\qquad\quad s.env[Y] \neq \bot
\end{array}\right.
\end{array}
$$

Figure 4.8: VC functions for the Getfield and GetfieldNullP rules

$$
\begin{array}{rcl}
VC_{\texttt{Putfield}}(P,X,F,Y) & = & \left\{\begin{array}{l}
\forall o \in \mathcal{Obj}, l \in \mathcal{L}, c \in \mathcal{Class}, \\
\quad [\![pre(P)]\!] \Rightarrow \\
\qquad s.env[X] = Loc(l) \Rightarrow \\
\qquad s.hp[l] = (c,o) \Rightarrow \\
\qquad\quad let\ v' = s.env[Y]\ in \\
\qquad\quad let\ o' = o[F \leftarrow v']\ in \\
\qquad\quad let\ p' = s.cpp^{+}\ in \\
\qquad\quad let\ hp' = s.hp[l \leftarrow (c,o')]\ in \\
\qquad\quad let\ s' = (s.env, hp', p')\ in \\
\qquad\qquad [\![post(P)[s'/s]]\!]
\end{array}\right. \\
VC^{NullP}_{\texttt{Putfield}}(P,X,F,Y) & = & \left\{\begin{array}{l}
\forall l \in \mathcal{L}, \\
\qquad pre(P) \Rightarrow \\
\qquad\quad s.env[X] \neq \bot
\end{array}\right.
\end{array}
$$

Figure 4.9: VC functions for the Putfield and PutfieldNullP rules

$$
[\![s \rhd_{C_0,M} (init, p_{end})]\!](Y, A_0, A_1) = \left\{\begin{array}{ll}
\exists l \in \mathcal{L}, c_{look} \in \mathcal{Class}, p_{beg} \in \mathcal{PP}, & \\
\qquad s.env[Y] = Loc(l) & \\
\wedge \quad s.hp[l] = (c,o) & \\
\wedge \quad c \preceq c_{look} \wedge c_{look} \preceq C_0 & \\
\wedge \quad \texttt{sig}(c_{look}, M) = (p_{beg}, p_{end}) & \\
\wedge \quad let\ v_0 = [\![A_0]\!](s)\ in & \\
\qquad let\ v_1 = [\![A_1]\!](s)\ in & \\
\qquad let\ e_t = s.env[\texttt{this} \leftarrow l]\ in & \\
\qquad let\ e_0 = e_t[\texttt{p}_0 \leftarrow v_0]\ in & \\
\qquad let\ e_1 = e_0[\texttt{p}_1 \leftarrow v_1]\ in & \\
\qquad\quad init = (e_1, s.hp, p_{beg}) &
\end{array}\right.
$$

Figure 4.10: Definition of the initial state of a method call

$s.env[Y] = l$. The class c_{look} is constrained by the literal $c \preceq c_{look} \wedge c_{look} \preceq C_0$ to be between c and C_0 in the class hierarchy, and required to implement the method by the literal $\texttt{sig}(c_{look}, M) = (p_{beg}, p_{end})$—which imply that $\texttt{sig}(c_{look}, M) \neq \bot$. This is not an exact axiomatising of the lookup algorithm, as it does not require c_{look} to be the *first* such super-class, but it does not matter as all implementations are required to enforce the same pre-condition (or a weaker one) and the same post-condition (or a stronger one).

Finally, the initial environment is built by giving the appropriate value to `this`, $\texttt{p}_0$ and p_1, the parameters of the method.

$$\begin{array}{l} \mathit{let}\ e_t = s.env[\texttt{this} \leftarrow l]\ \mathit{in} \\ \mathit{let}\ e_0 = e_t[\texttt{p}_0 \leftarrow v_0]\ \mathit{in} \\ \mathit{let}\ e_1 = e_0[\texttt{p}_1 \leftarrow v_1] \end{array}$$

and the entry point of the implementation selected is used as the program point of the initial state.

For the call instruction we have three semantic rules. We build one VC for each error rule and two VCs for the normal case rule (see Figure 4.11 on the following page).

The functions $VC^{no\ error}_{\texttt{Lookup}}$ and $VC^{NullP}_{\texttt{Call}}$ produce the VCs that ensure the absence of error. If the former is valid, then the pre-condition is sufficient to prove that the dynamic class of the object calling the method is a subclass of C_0. If the latter is valid, then the pre-condition is sufficient to prove that Y does not contain a null pointer.

The function $VC^{pre}_{\texttt{Call}}$ produces a VC ensuring that the call enforces the pre-condition of the method and that the initial state is well-formed. The function $VC^{post}_{\texttt{Call}}$ produce a VC ensuring that the method call enforce the post-condition of the current program point. As previously stated, the pre-condition and post-condition of the method called does not depend on the implementation returned by the lookup.

As for the VC for procedure call in the numerical core language, the VC quantify universally on the *end* state, but this time, as the initial state depends on the result of the lookup which can not be determined statically, both the pre-condition and the post-condition VCs quantify universally on the *init* state. Note that it is not necessary to quantify on both *init* and p_{end}, as once the implementation is fixed, both the initial state, the entry and exit point are chosen. But for the readability of the VCs, we used a shorthand for the lookup—the $s \rhd_{c_0,m} (\mathit{init}, p_{end})$ relation—which would be confusing without both quantification. An complete inlining of this shorthand would uncover a possible simplification, using the *let* ... *in* to fully define the *init* state.

As for the numerical language, the state *end* is not specified in terms of parts of *init*. The only link between those two states is the semantic invariant *compat* described in Definition 2.4.2 and the fact that *init* satisfies

$$VC^{pre}_{\texttt{Call}}(P,X,Y,C_0,M,A_0,A_1) = \begin{cases} \forall init \in \mathcal{State}, p_{end} \in \mathcal{PP}, \\ \quad [\![pre(P)]\!] \Rightarrow \\ \quad [\![s \rhd_{C_0,M} (init, p_{end})]\!](Y, A_0, A_1) \Rightarrow \\ \quad\quad [\![mth_pre(M, init)]\!] \wedge wf(init) \end{cases}$$

$$VC^{post}_{\texttt{Call}}(P,X,Y,C_0,M,A_0,A_1) = \begin{cases} \forall init, end \in \mathcal{State}^2, p_{end} \in \mathcal{PP}, \\ \quad [\![pre(P)]\!] \Rightarrow \\ \quad [\![s \rhd_{C_0,M} (init, p_{end})]\!](Y, A_0, A_1) \Rightarrow \\ \quad\quad [\![mth_pre(M, init)]\!] \Rightarrow \\ \quad\quad [\![mth_post(M, end)]\!] \Rightarrow \\ \quad\quad end.cpp = p_{end} \Rightarrow \\ \quad\quad wf(init) \Rightarrow \\ \quad\quad compat(init, end) \Rightarrow \\ \quad\quad let\ e = s.env[X \leftarrow end.env[\texttt{res}]]\ in \\ \quad\quad let\ s' = (e, end.hp, s.cpp^+)\ in \\ \quad\quad\quad [\![post(P)[s'/s]]\!] \end{cases}$$

$$VC^{no\ error}_{\texttt{Lookup}}(P,X,Y,C_0,M,A_0,A_1) = \begin{cases} \forall l \in \mathcal{L}, c \in \mathcal{Class}, o \in \mathcal{Obj}, \\ \quad [\![pre(P)]\!] \Rightarrow \\ \quad\quad s.env[Y] = l \Rightarrow \\ \quad\quad s.hp[l] = (c, o) \Rightarrow \\ \quad\quad c \preceq C_0 \end{cases}$$

$$VC^{NullP}_{\texttt{Call}}(P,X,Y,C,M,A_0,A_1) = \begin{cases} [\![pre(P)]\!] \Rightarrow \\ \quad s.env[Y] \neq \bot \end{cases}$$

Figure 4.11: VC functions for Call rule

$$VC_{\texttt{mth_pre}} = \begin{cases} \forall s \in \mathcal{State}, c \in \mathcal{Class}, m \in \mathcal{Method}, p_{beg}, p_{end} \in \mathcal{PP}, \\ \quad \texttt{sig}(c, m) = (p_{beg}, p_{end}) \Rightarrow \\ \quad mth_pre(m, s) \Rightarrow \\ \quad\quad pre[p_{beg}] \end{cases}$$

$$VC_{\texttt{mth_post}} = \begin{cases} \forall s \in \mathcal{State}, c \in \mathcal{Class}, m \in \mathcal{Method}, p_{beg}, p_{end} \in \mathcal{PP}, \\ \quad \texttt{sig}(c, m) = (p_{beg}, p_{end}) \Rightarrow \\ \quad pre[p_{end}] \Rightarrow \\ \quad\quad mth_post(m, s) \end{cases}$$

Figure 4.12: Global VCs

the pre-condition of the method and *end* satisfies the post-condition. This constraints must be sufficient to prove that the next state of execution, built using the environment before the call updated with the result of the call, enforces the post-condition of the program point P, where the call was made.

$$\begin{array}{l} \mathit{let}\ e = s.env[X \leftarrow end.env[\texttt{res}]]\ \mathit{in} \\ \mathit{let}\ s' = (e, end.hp, s.cpp^{+})\ \mathit{in} \\ \quad [\![post(P)[s'/s]]\!] \end{array}$$

Finally, Figure 4.12 on the previous page presents the VCs for linking the pre-conditions and post-condition of methods and of all their implementations. VC_{mth_pre} enforces that the pre-condition of a method is strong enough to ensure the pre-conditions of the entry points of all its implementations. VC_{mth_post} enforces that the pre-conditions of the exit points (which are annotated with *skip*) of all implementations of a method are strong enough to ensure the post-condition of the method.

Definition 4.4.2. We write $\mathcal{VC}_{obj}$ the VC calculus defined by the rules presented in Figures 4.8, 4.9, 4.11, and 4.12:

Let P be a program encoded as a flowchart and $b^\sharp$ be the result of an analyser encoded as *pre* and *post* predicates.

- If P is not well-formed, then $\mathit{false} \in \mathcal{VC}_{obj}(P, b^\sharp)$
- $\forall p \in \mathcal{PP}$, if $rule_i$ may apply on $\texttt{get_stmt}(p)$, then
$$\overline{VC}_{\texttt{rule}_\texttt{i}}(p, terms(\texttt{get_stmt}(\texttt{p}))) \in \mathcal{VC}_{obj}(P, b^\sharp)$$
- $\{VC_{continue}, VC_{\texttt{mth_pre}}, VC_{\texttt{mth_post}}\} \subset \mathcal{VC}_{obj}(P, b^\sharp)$

Theorem 4.4.2 (Soundness)**.**
The verification condition calculus $\mathcal{VC}_{obj}$ is sound:

Let P be a well-formed program and $b^\sharp$ be the untrusted result of an analysis. If all the verification conditions are valid

$$\forall \Phi \in \mathcal{VC}_{obj}(P, b^\sharp), \Phi\ \mathit{is\ valid}$$

the absence of run-time error is guaranteed by $b^\sharp$.

$$\forall s \in \mathcal{State}, e \in \mathcal{Err}, s \in \mathcal{Reach}(P) \Rightarrow s \not\rightsquigarrow e$$

Proof. Theorem 4.4.2 has been proved using the intermediate verification language WHY3 in the same way as Theorem 4.4.1 were:

- the VC calculus is specified as a set of lemmas parametrised by the implementation of a flowchart,
- the operational semantics of the language is specified as an interpreter,
- and the lemmas are used to discharge the proof obligations generated by the WP calculus.

More details can be found in Section 5.2. □

4.5 Conclusion

In this chapter we have adapted the deductive verification approach to the certification of the results of an analyser. We presented a general approach to generate a verification condition calculus in many-sorted first-order logic from a structural operational semantics.

The memory-model we defined is standard and, as in most deductive verification approaches, is a variant of the initial ideas of Cartwright and Oppen [CO81]. However, our approach took a step further by defining semantic states as a sort in its own right, which allows us to manipulate states in a functional way in VCs, thus the VCs follow closely the SOS. Approaches based on translation to an IVL define the memory model in a similar or even more precise way, but the state of execution is a global variable, and is modified by side-effects in all methods. A notable exception is the LOOP compiler [vdBJ01] which use a memory model accounting for the whole JAVA language, and translate java programs into *state transformers* in the logic of a proof-assistant. Therefore, the state of an execution is manipulated in a functional way during proofs, by application of state transformers. However, the aim of the LOOP compiler is to provide a framework to reason about JAVA programs, it provides tools to do proofs, not to automatise proofs. The result of the compilation is a series of theorems whose proofs are left to be done by the user in the expressive language of a proof-assistant, not a set of VCs expressed in a language well-suited for the ATPs in charge of discharging them, whereas our approach, when instantiated on a language and an analyser, provides a fully automatic result verification process.

To emphasise the simplicity of the relation between the VC calculus and the operational semantics, we present in Chapter 5 a practical approach to the justification of the VCs with respect to the operational semantics, and an experimental validation of the approach on numerical analyses. The same chapter presents our approach to the generation of VCs and establishes the soundness of the VCgen *w-r-t* the VC calculus.

By translating the abstraction of the program into pre-conditions and post-conditions, the VCs can manipulate the full memory, and we side-step the framing problem, so crucial to most deductive verification approaches (see Section 3.2.2). However, this does not solve entirely the problem as the main effort is moved on the ATPs, which may not be able to handle the VCs resulting from our approach. To address this concern—which involves primarily object-oriented programs as the experiments on numerical analyses show—we explain in Chapter 6 how to simplify the VCs depending on the abstract domain of analyses that are checked, and evaluate this simpler set of VCs on two object-oriented analyses.

Chapter 5

Experimentation in Why3: soundness of the VC calculus and numerical analyses result certification

Chapter 4 presented an encoding of memory models in many-sorted First-Order Logic (FOL) and a Verification Condition (VC) calculus derived from the operational semantics and the formalisation of an analyser as abstract interpretation. As discussed earlier, implementing a VC generator (VCgen) is not a trivial task and avoiding including it in the Trusted Computing Base (TCB) would require the certification of the VCgen. The present chapter proposes to generate VCs using the Weakest Precondition (WP) calculus of a standard Intermediate Verification Language (IVL, *e.g.*, Why3) and to establish the soundness of the VCs *w-r-t* the operational semantics implemented as an interpreter, inside the IVL. By doing so, the TCB of our approach includes a standard tool rather than a dedicated tool.

This is a good trade-off between the effort required to prove the implementation of the VCgen and the ability to adapt it to other languages and analyses, as it introduces some flexibility in the scheme: if the standard WP calculus is trustworthy, then it can be used to generate VCs for different languages and analyses. Moreover, recent work on the certification of the WP calculus of an IVL [HMM12, VJP10] clears the way to a fully certified VC generation. Remark that to really exclude from the TCB any dedicated tool, we can not rely on a compilation of programs targeting the IVL: we need to have in the IVL a representation of programs faithful to the programming language they are implemented in.

Based on this implementation scheme, some preliminary experiments were conducted to test the ability of off-the-shelf Automated Theorem Provers (ATPs) to discharge the generated VCs, first on an interval analysis, often

considered as the starting point for numerical analyses, and then on a polyhedral analysis, which is a more complex numerical analysis than the intervals as it provides relations between numerical variable.

A brief description of the WHY3 IVL is presented in Section 5.1. Then Section 5.2 details the implementation of a memory model as a theory, of a mostly small-step operational semantics as an interpreter, the specification of a VC calculus as lemmas, and how the proofs of Theorem 4.4.1 on page 65 and Theorem 4.4.2 on page 70 were obtained. Section 5.3 presents the results of the experimental validation of the methodology on numerical analyses. Finally, Section 5.4 discusses some related work and concludes.

5.1 The Why3 intermediate verification language

WHY3 [BFM+12] is the last iteration of the WHY [FM07] platform for deductive verification. It provides a ML like language for programming and specification, a WP calculus to generate VCs, and a back-end that allows the use of different ATPs (*e.g.*, Z3 [dMB08c], ALT-ERGO [CCK06], CVC [BT07], VERIT [BODF09], E [Sch02], VAMPIRE [RV99]) and proof assistants (*e.g.*, COQ [BC04]) to discharge the generated VCs. A WHYML file contains *theories* and *modules*, the first containing logical definitions and the latter containing programs and their specifications.

The base logic used in theories and specification is a variant of many-sorted FOL with algebraic data types and polymorphic functions. This logic happens to be—not completely incidentally—a super-set of the many-sorted logic used to define the theory of semantic states in Section 4.2.2. A theory is a list of declarations of types, functions, predicates, axioms and lemmas. Functions can be recursive or mutually recursive, but must terminate. Lemmas are formulae that will generate proof-obligations and will be added to the context of following proof-obligations, whereas axioms are not required to be proved. Using axioms, new data types can be axiomatised, *e.g.*, the theory of maps defines a new type `map` , two functions `get` and `set` , and the axioms mentioned in Section 4.2.1.

A module can contain any declaration valid in a theory, and defines programs and their specifications. A program can be non-terminating, contain loops, mutable field assignment, and raise exceptions. The specification of a program includes pre-condition, post-condition, variant (to prove termination), and effects (*e.g.*, which exceptions can be raised, which global values can be read, modified). Moreover, WHYML provides the statements `assert` ϕ, `assume` ϕ and `absurd`, with specific semantics, *i.e.*, rules for the WP calculus. *Assert*s are formulae that must be proved to hold, and that can be assumed to hold for the rest of the program, *assume*s on the other hand does not have to be proved to hold. The *absurd* keyword is used to cut paths that can not be reached in the control graph.

Lemmas and asserts can be useful to guide ATPs in the proof of a particular goal or proof-obligation. As the name implies, lemmas can be use to state simple (or simpler) intermediate properties that are easy (easier) to prove, and can be used in a more complex proof. In the same idea, asserts can be used to split the proof of a complex proof-obligation is several parts following the control flow of a program.

Once proof-obligations for programs and lemmas are generated, Why3 translate these formulae into the input syntax of different ATPs, *e.g.*, the SMT-LIB [STB10] syntax for SMT solvers, the TPTP syntax [Sut09] for ATPs, and Why3 provide means to define new translations. During these translation, Why3 applies transformations on the formulae to eliminate constructs that are not defined in the input syntax of the targeted tool, such as algebraic data type and polymorphic functions. But the transformations to apply are not limited to such eliminations, and can include arbitrary inlining of definition or splitting of goals at the choice of the user. These additional transformation, *i.e.*, not required to obtain formulae that conform to an ATP input syntax, can be be necessary for a proof-obligation to be discharged more quickly, or at all.

For details on the syntax of WhyML or on the features of the Why3 platform, the reader can refer to the Why3 manual [BFM+12]. Our experiments were done in the 0.74 version of the platform, but since they were conducted, a new iteration (the 0.80 version) has been released. Additional publications provide information on the translation of algebraic data types [Pas09] and polymorphic functions [BP11] or on the general architecture of the framework [BFMP11].

5.2 Relying on an Intermediate Verification Language for VC generation and VC soundness

The base logic of Why3 includes all the necessary construct to generate our VCs and prove a VC calculus sound:

- the terms of the syntax of the programming languages can be defined as an algebraic data-type,
- the flowchart description of a program can be given by functions over these types,
- the memory model of the language, *i.e.*, a theory of semantic states, can be defined using the theories provided by the Why3 standard library (*e.g.*, maps, records, tuples, integers),
- the translation of expression into terms of the logic—denoted by the $[\![\bullet]\!](\bullet)$ function—can be defined by functions in the logic of Why3, even for numerical expression, where recursion is necessary,

- a VC calculus $\mathcal{VC}_{gen}$ as defined in Definition 4.3.1 on page 60 can be specified by lemmas quantifying over program points. Informally, these *generalised lemmas* state that all program points valid the VCs.

As lemmas in WHY3 generate proof-obligations, the framework can be used as a high-level interface with ATPs: the VC calculus is specified in Many-Sorted First-Order Logic, and WHY3 generates the VC and send them to ATPs.

Moreover, the WHYML programming language can be used to write annotated programs over the theory of semantics states: programs and annotations can manipulate objects of types $\mathcal{State}$, $\mathcal{Heap}$, etc. Therefore, the operational semantics can be implemented as an interpreter, and the soundness of the VC calculus *w-r-t* that interpreter can be established, providing a mechanised proof of Theorem 4.4.1 and Theorem 4.4.2.

5.2.1 Overview of the implementation of the approach

The goal of the implementation in WHY3 is twofold: generating the VCs on a program, and establishing the soundness of the VC calculus *w-r-t* the operational semantics.

Given a program and the result of an analyser, the implementation in WHY3 can be used to generate VCs and discharge them with ATPs. It does not require any translation of the program into the WHY3 IVL, but is done by specifying the flowchart and by instantiating the syntactic domains. The specification of the flowchart is equivalent to providing the Abstract Syntax Tree of the program as a data type in WHY3, if the input programming language was structured. No translation or program transformation is required.

Given a VC calculus and an operational semantics, the implementation can be used to prove the VCs sound *w-r-t* the semantics, once and for all. This does not provide the same formal guaranties as a formal proof of a VCgen, because it requires to trust the implementation of the WP calculus of WHY3 and the implementation of the tools used to discharges the proof-obligations (which can be ATPs or a proof-assistant). However, those tools are used to discharge the VCs generated to certify the result of an analyser and belong to the TCB of the approach anyway.

Figure 5.1 on the next page presents an overview of the implementation and of its two uses: on right hand side, the certification of the result of an analyser on a program (see Section 5.2.2), and on the left hand side the certification of the soundness of the VC calculus (see Section 5.2.3). As can be seen on the figure, the definition of the syntax of the programming language, the theory of semantic states and the specification of the VC calculus are reused in both cases. The syntactic domains, the flowchart of the program and the abstraction calculated by the analyser are specified by

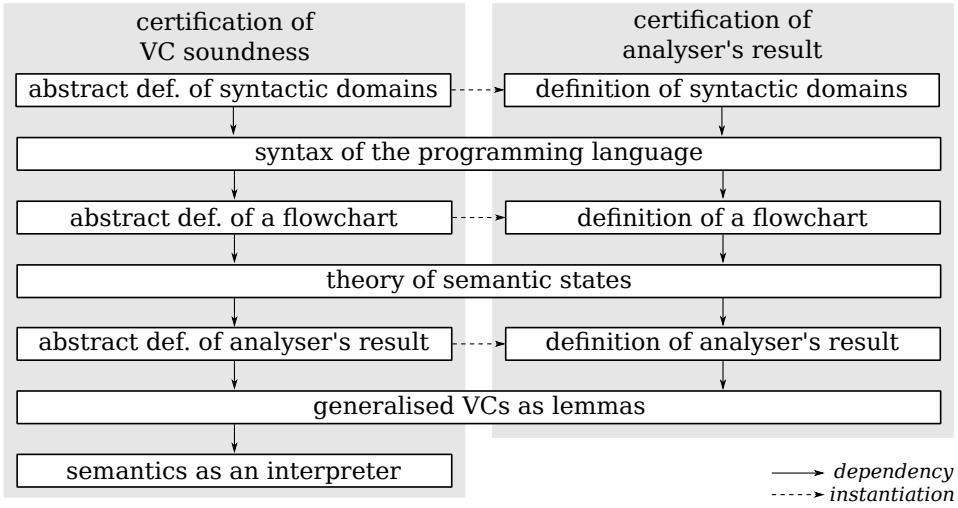


Figure 5.1: Overview of the implementation

abstract types and functions while proving the soundness of the VC calculus, and instantiated on a given program and the result of an analyser to generate VC. Note that the interpreter is only used to prove the soundness of the VC calculus.

type $\mathcal{PP}$	**type** $\mathcal{PP}$ = $\mathtt{PP}_0$
function p^+: $\mathcal{PP} \to \mathcal{PP}$	**function** p^+ = $\mathtt{PP}_0 \to \mathtt{PP}_0$
predicate pre: $\mathcal{PP} \times \mathcal{State}$	**predicate** pre p:$\mathcal{PP}$ s:$\mathcal{State}$ = ...

Figure 5.2: Abstract definition of an annotated flowchart vs specification of an annotated program

Figure 5.2 illustrates—using a *pseudo*-WHY3 syntax—the differences between, on the left hand side, the specification of a flowchart using *abstract definitions*, and on the right hand side, the specification of a program by *instantiating* the abstract definitions. In abstract definitions, the implementation of types, functions and predicates are not given, only the functional type of functions and the types of the arguments of predicates are specified. This does not prevent WHY3 from generating VCs, but the non-implemented functions and predicates will be treated as uninterpreted symbols by ATPs.

5.2.2 VC generation

The first step to generate the VCs on a given program and on the result of an analyser, is to instantiate the syntactic domains, *i.e.*, to define the types $\mathcal{PP}$, $\mathcal{Var}$, $\mathcal{Var}_{\mathcal{A}}$, $\mathcal{Method}$ if the program is in the core numerical language and $\mathcal{PP}$, $\mathcal{Var}$, $\mathcal{Method}$, $\mathcal{Class}$ and $\mathcal{F}$ if the program is in the core object-oriented

language. Then, the flowchart of the program must be describe by defining the functions `next_pp`, `get_stmt`, and the functions describing the signatures of procedures or methods, summarised in Table 2.1 and Table 2.2. The class hierarchy is defined by the relation *subclass* and axioms stating that its transitive and reflexive. Finally, the annotations of the program must be written as predicates according to Listing 5.1. Recall that *pre* and *post* are defined in Section 4.3.1 as functions from $\mathcal{PP}$ to annotations in $\mathcal{Assn}(s)$, *i.e.*, they are formulae with $s \in \mathcal{State}$ as sole free variable besides their parameter p, whereas `pre` and `post` are predicates defined in the syntax of WHY3.

```
predicate pre(p,s0) = pre(p)[s0/s]
predicate post(p,s0) = post(p)[s0/s]
```

Listing 5.1: Implementation of the interpreter's annotations

The lemmas used to specify the VC calculus are generalisation and not the instantiated VCs that would be generated by a VCgen: they state that "all program points valid the verification conditions". The general form of the lemmas is presented in Listing 5.2, where the x_j are variables typed according to the terms expected in $stmt_i$. For each sort of statement $stmt_i$, there is one lemma per VC function $\overline{VC}_{\mathtt{rule}_i^k}$ in the VC calculus (the overlined notation correspond to the closed formula), and one lemma per global VC. As all terms are variables, no reduction is possible.

```
lemma generalised_VC_stmt_i^k :
    ∀p ∈ PP, ∀x_1 : sort_1, ..., x_n : sort_n,
        get_stmt(p) = stmt_i(x_1, ..., x_n) ⟹ VC̄_rule_i^k(p, x_1, ..., x_n)
```

Listing 5.2: Lemma specifying the VC calculus

While certifying the results of an analyser on a program, these lemmas need to be proved on the implementation of the functions describing the program and the predicates describing the analyser's result. At this point, the VCs defined in the Chapter 4 are simply projections of the generalised lemmas on each program point. These projections can be obtained either by requiring WHY3 to apply transformations—*i.e.*, inlining and splitting—on the lemmas while outputting the proof-obligation to an ATP, or by generating automatically intermediate lemmas using an untrusted external program. These intermediate lemmas can then be used by ATPs to discharge the generalised lemma, ensuring that a potential error while generating the projection does not jeopardise the result certification. Both solutions—projections using transformations of proof-obligations or as intermediary lemmas—are tested during the experiment on numerical analyses, and a comparison is done in Section 5.3.3.

```
(* instantiation of syntactic domains, flowchart and annotations*)
type 𝒫𝒫 = PP0
type 𝒱ar = X
function p+ = PP0 → PP0
function get_stmt p:𝒫𝒫 = PP0 → X := X+1
predicate pre p:𝒫𝒫 s:𝒮tate = s.env[X] ≥0
predicate post p:𝒫𝒫 s:𝒮tate = s.env[X] > 0

(* projection on program point PP0 : VC(PP0)*)
lemma projection_PP0 :
   ∀s ∈ 𝒮tate,
      pre(PP0, s) →
         let e′ = s.env[ X <- s.env[X]+1 ] in
         let s′ = (e′,s.ars, PP0) in
            post(PP0,s′)

(* generalised lemma for assignment *)
lemma gen_VC_assign :
   ∀p ∈ 𝒫𝒫, x ∈ 𝒱ar, e ∈ ℰxpr,
       get_stmt(p)= x := e →
       ∀s ∈ 𝒮tate,
          pre(p,s) →
             let p′ = s.cpp+ in
             let env′ = s.env[x <- ⟦e⟧(s) ] in
             let s′ = (env′ , s.ars , p′ ) in
               post(p,s′)
```

Listing 5.3: Example of an instantiation with intermediary lemma

Listing 5.3 details the example of the instantiation of the necessary types and functions on a simplistic program with only one program point $\mathtt{PP}_0$ annotated with an assignment, and the lemmas relevant to that statement. The `gen_VC_assign` lemma is the generalised lemma for the assignment: it follows the structure given in Listing 5.2 and uses the $VC_{\mathtt{Assign}}$ function defined in Section 4.4.1. The `projection_PP` $_0$ lemma is a projection of the generalised lemma specifying the VC on the program point $\mathtt{PP}_0$. This projection instantiate the variable X, the expression $X := X + 1$, and the next program point $\mathtt{PP}_0$, thus the quantification over instantiated variables and the literal `get_stmt(`p`)=` x `:=` e have been eliminated. We used in this listing the same notations than in the previous chapters, but some do not belong to the syntax of WHY3 and are replaced by functions in the implementation.

5.2.3 Small step semantics as an interpreter in the IVL

To establish the soundness of the VC calculus, the operational semantics is implemented as an interpreter in the WHYML language. The syntactic domain, the functions implementing a flowchart, and the predicates imple-

menting its annotations, are left abstract. The generalised lemmas specifying the VC calculus are supposed valid—*i.e.*, the VC are supposed true on all program points—and the WP calculus of WHY3 is run on the interpreter. The generated proof-obligations are then discharged either by an ATP or in a proof-assistant (in fact, in this case all VCs were discharge by ATPs). They establish the soundness in the sense that, if the VC are valid on a program and its annotations, the interpreter never run into an error state, *i.e.*, the program never reach an error state. This definition of soundness corresponds to Definition 4.3.2 if the interpreter is a faithful representation of the semantics, and under this condition, WHY3 can be used to prove Theorem 4.4.1 and Theorem 4.4.2.

The semantic relations $\longrightarrow$ can be modelled by interpreters implementing a function `eval` : $\mathcal{State} \rightarrow \mathcal{State}$. The interpreters can be annotated in the IVL with pre-conditions and post-conditions, which correspond to what the VC calculus certifies: for each program, for each program point, if the annotated pre-condition is satisfied by the execution state on entry, then the execution state on exit satisfies the annotated post-condition, and no error state is reached.

To implement the operational semantics, an interpreter do a case analysis on the statement annotating the current program point. The pattern followed by the implementation of the interpreters is presented in Listing 5.4. Note that the annotations of the interpreters are abstractions of the program's pre-conditions and post-conditions. When describing the semantics of a language with an interpreter the annotations of a specific program are not known and are abstracted by the two unimplemented predicates `pre` and `post` .

```
let eval (s:State) =
{ pre(s.cpp,s) }
   match get_stmt s.cpp with
      | stmt1  →
      | stmt2  →
      ...
   end
{ post(s.cpp,result) }
```

Listing 5.4: Pattern used for the interpreters

Transitive closure. To account for the relation $\longrightarrow^*$, transitive and reflexive closure of $\longrightarrow$, the function `eval` is recursive and mutually defined with a function `eval_closure`, which simply iterate `eval` until the exit program point is reached. The closure asks for its arguments to validate the precondition of the method/procedure called, therefore it needs to have the method and class names or the procedure name as arguments.

```
let eval_closure (c:Class) (m:Method) (init:State) (s:State)=
{ mth_pre(c,m,init) ∧ pre(s.cpp,s) ∧ sig c m ≠ None }
   match sig c m with
      | None  →  absurd;
      | Some (p_beg, p_end)  →
          if s.cpp = p_end
          then s
          else
              eval_closure c m init (eval s)
  end
{ mth_post(c,m,result) }
```

Listing 5.5: Transitive closure of the eval function

The iteration is performed by a call on `eval` and a recursive call to `eval_closure`, presented in Listing 5.5. Therefore, the closure is not only called on the entry point of methods but also on intermediary states, and the annotations of the closure must be implied by the pre-condition of the methods, but the pre-conditions of the intermediary states must also holds during the iteration. To account for both situation, the closure has two states as arguments: the initial state of the call `init` and the current state of execution `s` . The pre-condition of the closure is therefore that the pre-condition of the method called holds on the initial states—*i.e.*, $\texttt{mth_pre}(\texttt{c}, \texttt{m}, \texttt{init})$ holds—and that the pre-condition of the current states of execution holds—*i.e.*, $\texttt{pre}(\texttt{s.cpp}, \texttt{s})$ holds. The last part of the pre-condition, $\texttt{sig}(\texttt{c}, \texttt{m}) \neq \texttt{None}$ is simply stating that the class `c` and method `m` correspond to an actual implementation. It is already established during the lookup, but we need to include it in the pre-condition to discard the possibility that no exit program point could be found.[1]

Ill-formed programs. To discard ill-formed program, we demand a proof that all program points are syntactically well-formed by generating corresponding proof-obligations. As the syntax is described as a sort of the logic, a predicate that check syntactic properties can be defined. To generate a proof-obligation, we add two simple lemmas, `well_formed_stmt` and `well_formed_sig` , detailed in Listing 5.6 on the following page. The former checks that parameters are never assigned and the latter checks that all call specify a class that implement the method.

Error cases. As in the semantics, *error cases*—*e.g.*, array out of bound, null-pointer dereferencing—can be treated in several ways. In the semantic world, they can be treated by the addition of special error states—that

[1] An alternative would be to push the exit program point among the argument of the closure, but as the current implementation lead to a trivial proof-obligation anyway, we preferred to keep the arguments of the closure simple.

```
lemma well_formed_stmt : ∀ p:pp . well_formed_stmt p
lemma well_formed_sig :
   ∀ p:pp, x y:var, cmax:class, m:method, arg1 arg2:var.
       get_stmt p = Call x y cmax m arg1 arg2  →
           not (sig cmax m = None)
```

Listing 5.6: Well-formedness of programs

can not be reduced further—or the semantics can simply block. In the interpreter, these choices can be translated as either doing checks in the control graph of the interpreter and raising an exception/including a error state in the type $\mathcal{State}$, or calling semantic functions with pre-conditions. The former will be illustrated in Section 6.3.1 to deal with *NullPointer* errors, while the latter will be illustrated in Section 5.3.1 to deal with *array out-of-bound* errors.

Another option to model a blocking semantics would be to use the *absurd* keyword in the branches of the interpreter that lead to a blocked execution. It generates a proof-obligation that entails proving the keyword is never reached, therefore proving the semantics of a program is never blocked. It is similar to the current settings, that uses pre-condition of auxiliary function to generate these proof-obligations.

Variant of operational semantics. The semantics used in the proof-of-concept are deterministic, but we believe that it is not a requirement of the approach. To account for an non-deterministic semantics, a choice operator could be axiomatised in Why3 in the same way a random generator is—when no assumption on probability distribution is made. A simple function `choice` could be specified, with the only requirement that it result should belong to a particular set of values. The result in proof obligations would be the introduction of a disjunction upon the possible value returned by the call to `choice` .

5.3 First experiment: result certification of numerical analyses

To describe the results of our approach, we first instantiate the scheme on two numerical analyses performed on different programs. To obtain the invariants, we used a web demo provided by Bertrand Jeannet [Jea], using the *box* (for the intervals) and *convex polyhedral (polka)* abstract domains. All development can be found at `http://www.irisa.fr/celtique/ext/chk-sa-boogie`.

5.3.1 Numerical language VCgen

For the core numerical language, array out of bound errors block the semantics. Therefore in the definition of the `eval` function of the interpreter, accesses and updates of arrays are done by auxiliary functions `set_array` and `get_array` as defined in Listing 5.7. They have their own pre-condition—which enforces the necessary check on the bounds of the array—and post-condition—which reflects the semantics of the operator.

```
let get_array (a:array) (i:int) =
{ 0 ≤ i < a.len}
      M.get (a.elts) i
{result = a.elts[i]}
let set_array (a:array) (i:int) (v:int) =
{0 ≤ i < a.len}
     ( len = a.len ; elts = M.set a.elts i v )
{result = ( len = a.len ; elts = a.elts[i<-v] )}
```

Listing 5.7: Functions manipulating arrays

5.3.2 Experiment

All VCs, even the proof-obligation generated on the interpreter to prove the soundness of the VC calculus, were discharged by a combination of SMT solvers—Alt-Ergo (version 0.94), CVC3 (version 2.4.1) and Z3 (version 2.2)—on a 3 years old laptop equipped with 4 GiB of memory and a 2.93 GHz Intel Core 2 Duo and using Linux. We chose SMT solvers for this first experiment on numerical analysis because they integrate powerful arithmetic decision procedures. The tests were done on a factorial program and a binary search program[2].

The standard binary search program makes an interesting case study for a relational analysis, as it involves multiples array accesses at indexes between bounds indirectly related to the array length. As stated in Section 4.4.1, array bound checks are enforced in the interpreter through the use of array access functions, and appear in the corresponding axiomatising theorems. If the analysis is precise enough, discovered invariants should be strong enough to prove the safety of array accesses.

To illustrate the elimination of explicit array bound checks, the analyses not only deals with integer variables, but also takes into account the length of the arrays. However to prove the absence of error in the binary search program requires a relational analysis[3], thus the interval analysis was only

[2]In order for the analyser to return meaningful results, the multiplication was implemented as a loop and the division by two was integrated as an additional operator

[3]As parameters of procedures are not reassigned, the notion of relational analysis extend to relations between the arguments of a procedure call and its result.

ran on the factorial program.

In the binary search program an integer variable `s` is used as a surrogate Boolean variable, with value `1` and `0` standing for `true` and `false`, not intended to be related to the other variables. Nevertheless, the analyser used for this experiment discovered hidden but unnecessary relations between all the integer variables, and invariants tended to be very large, with lots of noise. This is not unusual but rather inevitable for static analysers and a certification scheme should be robust regarding such convoluted invariants. The interesting point is that those invariants were checked and were precise enough to ensure array access safety. Even if the analysis results were far from invariants used in manual program proof, no post-processing was necessary to prove the corresponding VCs.

5.3.3 Results

In a first time, the ATPs were launched directly on the generalised lemmas specifying the VC calculus described in Listing 5.2 on page 77. However, some VCs were not discharged by any solver. To complete the proof, we could either instantiate the generalised lemma on some program points, and obtain a new lemma, effectively stating a VC to be proved, or split the unproved goal using WHY3 transformation.

The first solution—illustrated in Table 5.1 on the following page—involves generating new auxiliary lemmas. As argued before, this can be done automatically with no loss of trust in the final result as long as the generalised lemma specifying the VC calculus is proved. Moreover, this solution allows selective projection of the unproved goal only on the relevant program points, which limit the number of generated auxiliary lemmas.

Recall that each generalised lemma quantify over all program points, thus the generalised lemma for assignment, for instance, need only to be projected on program points annotated by assignment. We call these program points *relevant* to this particular lemma. Projecting on program points annotated with any other statement leads to trivial lemma of little interest for ATPs.

In the given example, to prove the generalised $VC^{pre}_{\texttt{Call}}$, 3 auxiliary lemmas are used. Two are the VC for the relevant program points—*i.e.*, p_4 and p_5 in this case—and one is a subgoals of the VC for p_5, and correspond to an inlining of the *post* function. Alt-Ergo discharge the auxiliary lemma and the projected VCs whereas Z3 and CVC3 reach timeout (as indicated by the " ! "), but the exact opposite happens for the generalised lemma.

The second solution, illustrated in Table 5.2 on the next page, is automatic but leaves the split to a tool oblivious to the VC generation methodology and to the structure of the VCs. This lead to a large number of subgoals, as the transformation requires both inlining and split, and do not typically result in one subgoals per program point. In any case, subgoals will be generated for irrelevant program points, but the reasons for this

Proof obligations	Alt-Ergo(0.94)	CVC3(2.4.1)	Z3(2.2)
$VC^{pre}_{\texttt{Call}}(p_4, \ldots)$	0.07	!	!
$VC^{pre}_{\texttt{Call}}(p_5, \ldots)_{aux}$	16.91	!	!
$VC^{pre}_{\texttt{Call}}(p_5, \ldots)$	0.72	!	!
$VC^{pre}_{\texttt{Call}}$	!	8.45	0.18

Table 5.1: Result for $VC^{pre}_{\texttt{Call}}$ on polyhedral analysis of factorial program

irrelevancy—typically, the VC does not concern the statement annotating a program point—may be harder to detect after inlining.

Proof obligations	Alt-Ergo(0.94)	CVC3(2.4.1)	Z3(2.2)
$VC^{post}_{\texttt{Call}}$			
$Compat(s, s')$	3.98	!	!
$post(p, s')$			
subgoal$_1$ p_0	0.09	!	!
subgoal$_2$ p_1	0.04	0.72	!
subgoal$_3$ p_1	0.06	!	!
...			
subgoal$_8$ p_4	0.14	0.74	!
subgoal$_9$ p_4	17.90	!	!
subgoal$_{10}$ p_5	0.15	0.73	!
subgoal$_{11}$ p_5	5.22	!	!
...			

Table 5.2: Result for $VC^{post}_{\texttt{Call}}$ on polyhedral analysis of factorial program

In the given example—a proof of the generalised $VC^{post}_{\texttt{Call}}$ for the factorial program and the polyhedral analysis—a first split isolate two subgoals, $VC^{post}_{\texttt{Call}}$ and $post(p, s')$. The former is proved by Alt-Ergo while the latter has to be inlined and split and result into 27 subgoals, each related to a program point between p_0 and p_{14}—some program points being related to two subgoals (p_1 for example) while others are related to only one (as for p_0) depending on the shape of the post-condition. We did not show all subgoals in the table to keep it readable but left all relevant parts to keep it informative. The only program point where a call do occur are p_4 and p_5, which explains the significantly longer time taken by Alt-Ergo to prove the corresponding subgoals. Nevertheless, even subgoals associated with irrelevant program point were not discharged by Z3 or CVC3.

Overall, only Alt-Ergo and Z3 are necessary to prove all subgoals, but the goals proved by CVC3 were not all proved by one of the others. For practical issues, we set a timeout of 30s on *subgoals*, not on *goals*. As a

matter of fact, the overall time taken by Alt-Ergo to prove $VC^{post}_{\texttt{Call}}$ amounts to 26.62s, but such numbers are difficult to compare to time spent on goals discharged with neither auxiliary lemma nor split/inline transformations. In one case a number of completely separated proofs are done, whereas most of the context is the same, in the other, the context is given once. Besides, a timeout result on a lemma proved directly by one prover may be unfair to the other provers, as a split may change the result and entails, in a sense, a much larger timeout distributed over all subgoals.

Our experiment can not be taken as a benchmark to establish the superiority of one solver over the others. On the contrary, it demonstrates that the possibility to use different ATPs can be useful to prove a given set of VCs, or even a particular one. In the two previous examples, results seem to indicate that Alt-Ergo was the only solver able to discharge most subgoals. However, the differences between the two examples, *i.e.*, Z3 and CVC3 having a hard time on the fully split and inlined goal for $VC^{post}_{\texttt{Call}}$ but being the only ones to discharge $VC^{pre}_{\texttt{Call}}$ (and pretty quickly for Z3), indicate that they were able to handle the generalised lemma specifying the VC calculus and use the auxiliary projected lemmas efficiently.

Most subgoals (92% of 94 subgoals) were proved in less than 1s by one of the SMT solvers. Z3 for instance typically either reaches timeout or is very fast—and actually never takes more than 2s when it succeed—even on subgoals on which the other solvers reaches timeout or take significantly longer than *usual*. On the contrary, Alt-Ergo and CVC3 exhibit a wide range of behaviours, with subgoals proved very quickly—even some on which Z3 reaches timeout—and others (but very few) subgoals proved in 15s.

Altogether, the three different provers behaved very differently. Each solver has a specific input standard, and the WHY3 back-end performs different transformations to encode its high level features in each standard, therefore it is difficult to determine the reasons behind these differences. What the experiment has shown is that one solver may not be enough to achieve full automation. This fact may have dire consequence when considering the TCB of a result certification solution as any new solver is a new source of errors, and we examine a partial solution for excluding ATPs from the TCB in Chapter 8.

5.4 Conclusion and related work

The present chapter detailed an approach to generate VC and establish their soundness with-respect-to the operational semantics using an intermediate verification language and its weakest pre-condition calculus, and presented some experimental results.

5.4.1 Trusted Computing Base

The soundness of this approach relies on the adequate modelling of the language, and on the Weakest Precondition calculus (WP) of Why3. To prove a particular analysis result on a specific program, one has only to prove the VCs generated by the WP calculus on the specification of the VC calculus in Why3 establishing point-wise correctness of the result of an analysis on a particular program. Hence the first part of the Trusted Computing Base (TCB) is the Why3 tool and its WP calculus.

Programs are not transformed to an Intermediate Verification Language (IVL), as their representation is explicit, rather, the semantics of the language is described in the IVL of Why3 and program invariants (or conditions) are described using the logic provided in the IVL. No transformation on top of the IVL is required, hence no compiler is added to the TCB.

The analysis result must be embedded in the logic on the IVL. This can be done either by translating the concretisation in a predicate over the abstract domain or providing directly the *pre* and *post* predicates used in the verification conditions—which amount to a partial evaluation of the translated concretisation function on the abstraction of the program. Therefore if the aim of the process is to ensure the soundness of the abstraction calculated by the analysis, this translation from the abstract domain to assertions is part of the TCB. However, if the result of the analysis is understood as a verdict on the absence of errors at run-time in the program, then the faulty translation of an unsound abstraction does not matter: as long as the verification conditions are valid, the program is proved safe. In that case the translation of abstractions to assertions is not part of the TCB.

Goals are discharged using Coq or automatic solvers after translation from the Why3 syntax and application of specific transformation to eliminate high-level constructs and simplify or split the goals. The TCB depends on the automatic and interactive tools used to make the proof, and on the Why3 tools in charge of the translations and transformations. The use of an interactive proof assistant is acceptable when doing the interpreter's proof, as it needs to be done only once; but in a certification scheme, the proof of goals specific to a program should be done automatically, to make sure that no knowledge of the analysis is required from users and that we do not include the analysis in the TCB. On the other hand, Why3 provides no facility for checking automatic provers results for now and the TCB includes these rather complex tools.

With respect to deductive verification approaches, our approach is a middle-ground between a compilation approach (*e.g.*, Krakatoa [MPMU04], Spec# [BLS04], VCC [CDH+09]), whose TCB includes a compiler performing transformations and typing, and full program proofs using a framework dedicated to a single language (*e.g.*, KeY [ERB+06]), whose TCB only includes the WP calculus, but are restricted to programs written in one in-

put language (*e.g.*, JAVA for the KEY platform). In both cases, automatic provers are widely used to discharge verification conditions, and must be added to the TCB.

The TCB of our approach includes tools—*i.e.*, an IVL and ATPs—that are not deemed worthy of trust in foundational proof-carrying code approaches [App01, CCNS05, AAV02, WNKN04]. However, we rely on an IVL to generate the VCs but also to justify the VC calculus, as the same generalised lemmas are responsible for the generation of VCs and proved *w-r-t* the operational semantics. This provides a *declarative* implementation of the VCgen: one has only to specify the VC calculus, a program, and the result of the analyser, to generate VCs. In a sense, we use WHY3 as a VCgen generator, and the soundness of the generation of VCs is dependent on the encoding of algebraic data types and recursive functions into many-sorted FOL.

The proof of the soundness of the VC calculus, which involves implementing the operational semantics of a language as an interpreter, is not a foundational proof, in that it rely on the WP calculus implemented in the WHY3 platform. But it is a middle-ground between a foundational approach and no guaranties, and it allows to prove the VC calculus and the VCgen at the same time. Furthermore, providing a foundational proof for verification frameworks based on IVL is an active field of research [HMM12, VJP10], and so is providing foundational proofs for state-of-the-art ATPs [BW10, AFG+11, BCP11, Mos08, FMM+06]. Such approaches could in the short or medium term eliminate some of the concerns regarding the TCB of our approach.

5.4.2 Certified static analysis

Albert *et al.* [APH05] have developed an Abstract-Carrying Code approach, that attach to programs the abstraction calculated by an analyser. The analyser is a CLP abstract interpreter that calculate a fix-point sufficient to ensure a safety policy, and the verification of the fix-point iterate the CLP program one time. The TCB of such a PCC framework includes the analyser—which in this case contains the specification of the analyser and the CLP engine—whereas our approach aims specifically at removing the analyser from the TCB.

A foundational proof of safety for a static analyser can be obtained by certifying the analyser inside a proof-assistant. Klein and Nipkow have formalised the Java byte code verifier in ISABELLE [KN03], and Pichardie *et al.* [CJPR05, Pic05] formalised the abstract interpretation framework [CC77] in COQ and used it to prove the soundness of several program analysers. This last approach requires to develop and prove in COQ the whole analyser which is a formidable effort of certification and raises efficiency concerns, COQ being a pure lambda-calculus language. Another way to obtain a

foundational proof of safety is to certify, inside the proof-checker, a verifier of analysis result rather than the analyser. Besson *et al.* [BJPT10] applied this *result certification methodology* [WB97] to a polyhedral analysis, developing an analyser together with a dedicated checker whose soundness is proved inside Coq, but which is limited to linear invariants, whereas we can certify any result the ATP can handle.

The work closest to our approach is the result verification of resource analyses proposed by Albert *et al.* [ABG+11] who have shown how results of the state-of-the-art static analysis system COSTA can be checked using the verification tool KeY. COSTA produces guarantees on how resources are used in programs. Resource guarantees are expressed as upper bounds on number of iterations and worst-case estimation of resource usage, and injected into KeY as JML annotations. The derived proof obligations are proved automatically using the prover of KeY. Our approach defines a methodology applicable to a wide range of analyses, and generates VC using a standard IVL rather than relying on a framework dedicated to a particular language, such as KeY. Overall, we provide a more general framework that use standard tools and can rely on multiple provers, but we have only developed a proof-of-concept, whereas they focused on a particular analysis and a particular tool to achieve a complete integration.

5.4.3 Limits

If the soundness of the approach is formally proved, its completeness is not clearly established, as the verifications conditions may not be appropriate for all analysers. They are sufficient to prove program safety, as the proof of the interpreter established, but they may not be necessary, and may be too strong to account for some abstract domains. Moreover the amount of automation is a concern. The statement of the semantics of the language and the verifications conditions were carefully crafted, as minor syntactic differences can make a huge difference when discharging the verification conditions. A sound representation, easily proved in Coq using a limited set of tactics in a systematic way, may prove very challenging for ATPs. On the other hand, large invariants may involve simple but fastidious reasoning in Coq, whereas ATPs performed very well. Using auxiliary lemmas in an IVL to cut proofs in parts of different complexity could be an approach to get the best of both worlds.

Chapter 6

Verification Condition simplification for analyses of the heap

The time spent by Automated Theorem Provers on some VCs is source of concern, not as such for the result certification of numerical analysis, but when dealing with more different kind of abstract domains, *e.g.*, the BCV or the null pointer domain. Contrary to numerical analyses, the verification conditions for even the simplest object-oriented program analyses are neither quantifier-free, nor do they fall into existing decidable fragments. Our experiments show that for now, ATPs are usually incapable of discharging such verification conditions. To circumvent this obstacle, we suggest to restrict to a particular class of analyses in order to obtain a more parsimonious embedding of abstract domains into pre-conditions and post-conditions. We will use the same methodology as employed in Chapter 5 to generate VCs with a new specialised VC calculus dedicated.

Section 6.1 presents a family of object-oriented analyses, described by a parametrisation of the instrumentation of the semantics and of a generic abstract domain. Section 6.2 details a set of quantifier-free verification conditions tailored to the family of analyses previously described. Section 6.3 evaluates the methodology on two object oriented analyses: a simple, classical analysis (the BCV), and a more complex analysis that requires instrumentation of the semantics (the null pointer analysis). Finally, Section 6.4 concludes the present chapter and presents further research directions to explore.

6.1 Restriction to a family of analyses

The VCs presented in Chapter 4 were derived from the semantics and are very general. To be simplified, the VC calculus must be less general, which

entails restricting the number of analyses whose results can be certified. Rather than choosing a particular analysis and defining a dedicated VC calculus, we define a new VC calculus restricted to a *family* of analyses: analyses that have comparable concretisation functions, as defined in Section 6.1.2, and that use the same kind of instrumentation of the semantics, as defined below.

6.1.1 Instrumented semantics

The soundness of an analysis is given *w-r-t* the concrete semantics of the language. For a good number of analyses, only the set of reachable states is needed to state correctness. However, some properties on the execution of a program may be hard or impossible to describe on the reachable states alone as some information on the trace of the execution may have been lost. For example, the semantics of an undefined field is that it contain null, hence it can not be distinguished from a field that has been initialised with null if the only information available is the reachable states, whereas it can be observed in the complete trace of execution.

To add some information from the trace of execution to the set of reachable states, a common approach is to *instrument* the semantics, *i.e.*, to add to semantic states some form of *flag*. These flags should not change the semantics of a program, hence they can be modified but can not influence the evolution of all non-instrumented parts of the state. The *instrumented semantics* obtained can be projected on the normal semantics by forgetting all information about the flags.

For the analyses we want to certify, the *initialisation* of an object is essential, and needs to be monitored. We add a flag to the fields, that will have different meanings depending on the analysis we want to certify. This flag is modified when an object is allocated and when the field is updated. Different analyses may require a different instrumentation, but the approach is always the same. To retain some generality of the VC calculus, we define an instrumented semantics parametrised by the domain of flags $\mathcal{IF}$, the default flag *inull*—used when a field is set to null during the allocation of an object—and the function *ifield*—used to update the flag of a field. The domain $\mathcal{Obj}$ of objects is updated to include the instrumentation of fields.

$$\mathcal{Obj} = \mathcal{Class} \times (\mathcal{F} \rightarrow \mathcal{Val} \times \mathcal{IF})$$

Given an object o and a field f, we write $o(f)_1$ (respectively $o(f)_2$) the projection of $o(f)$ on the value (respectively the instrumentation).

To define the two analyses that concern us the function *ifield* only has to take as argument the flag to be updated, but this could be generalised, and any information available while applying the update field semantics rule could be of interest, *e.g.*, the dynamic class of the object containing the field, the type of the value assigned, or the current program point. Figure 6.1 only

details the rules for allocation of an object and for update in a field, as all other rules only propagate the flags and leave them unchanged. The parts of the rules that have changed are emphasised using a bold font and underlined, to be compared with Figure 2.6 in Section 2.4.2.

$$\text{New}\ \frac{\begin{array}{cc} \mathit{get_stmt}(s.cpp) = x := \mathit{new}\ c & x\ \mathit{is\ assignable} \\ s.hp[l] = \bot & o = \lambda f.(\mathit{null}, \underline{\mathbf{inull}}) \end{array}}{s \longrightarrow (s.env[x \leftarrow l]\ ,\ s.hp[l \leftarrow (c,o)]\ ,\ s.cpp^{+})}$$

$$\text{Putfield}\ \frac{\begin{array}{ccc} \mathit{get_stmt}(s.cpp) = x.f := y & s.env[x] = l & s.hp[l] = (c,o) \\ v' = s.env[y] \quad (v, \underline{\mathbf{i}}) = o[f] & \underline{\mathbf{i}' = \mathbf{ifield}(\mathbf{i})} & o' = o[f \leftarrow (v', \underline{\mathbf{i}'})] \end{array}}{s \longrightarrow (s.env\ ,\ s.hp[l \leftarrow (c,o')]\ ,\ s.cpp^{+})}$$

Figure 6.1: Main differences between instrumented and non-instrumented semantics

The parametrisation of the instrumentation leads to a new semantic invariant: the flag of a field can not be modified "manually", it is initially *inull* and is only modified by successive application of *ifield*. This property is similar to the compatibility property we defined on the non-instrumented semantics, and leads to the definition of an updated $\mathit{Compat}_{I\mathcal{F}}$ predicate that replaces for all purposes the predicate *Compat*.

Definition 6.1.1. A state $s = (e, h, p)$ and a state $s' = (e', h', p')$ are compatible ($\mathit{Compat}_{I\mathcal{F}}(s, s')$) with the instrumented semantic relation if and only if

1. s and s' are compatible with the semantic relation

$$\forall l.l \in \mathit{dom}(h) \Rightarrow l \in \mathit{dom}(h') \tag{1}$$
$$\forall c, c', l, o, o'.h(l) = (c, o) \Rightarrow h'(l) = (c, o') \Rightarrow c = c' \tag{2}$$
$$e(\texttt{this}) = e'(\texttt{this}) \wedge e(\texttt{p}_0) = e'(\texttt{p}_0) \wedge e(\texttt{p}_1) = e'(\texttt{p}_1) \tag{3}$$

2. A field instrumentation can only be updated by iterating the *ifield* function (any number of time)

$$\begin{array}{l} \forall c, l, f, o, o'. \\ \quad h(l) = (c, o) \wedge h'(l) = (c, o') \Rightarrow \exists n.\ o'(f)_2 = \mathit{ifield}^n(o(f)_2) \end{array} \tag{4}$$

This definition completes Definition 2.4.2, and Proposition 2.4.1 holds for this extended compatibility relation. However, the iteration of the function *ifield* is not a construct of many-sorted FOL, and needs to be axiomatised.

$$\begin{array}{ll} \forall i \in \mathit{ifield}, & \mathit{ifield}^0(i) = i \\ \forall i \in \mathit{ifield}, n \in \mathbb{N}, & \mathit{ifield}^n(i) = \mathit{ifield}(\mathit{ifield}^{n-1}(i)) \end{array}$$

6.1.2 Parametrised analyses

We restrict our attention to analyses parametrised by an abstract domain $\mathcal{V}al^\sharp$. A variable `x` is abstracted in a flow-sensitive manner by an element $v \in \mathcal{V}al^\sharp$; a field f is abstracted in a flow-insensitive manner by a pair $(v_1, v_2) \in \mathcal{V}al^\sharp \times \mathcal{V}al^\sharp$ such that v_2 is the abstraction of `x.f` providing v_1 is the abstraction of `x`. The form of the abstract domain is defined by

$$\mathcal{H}eap^\sharp = \mathcal{F} \to \mathcal{V}al^\sharp \times \mathcal{V}al^\sharp \qquad \mathcal{E}nv^\sharp = \mathcal{V}ar \to \mathcal{V}al^\sharp$$
$$Abs = \mathcal{H}eap^\sharp \times (\mathcal{PP} \to \mathcal{E}nv^\sharp)$$

The concretisation function $\gamma_{\mathcal{R}each}$ is parametrised by a set $\gamma_{null} \subseteq \mathcal{V}al^\sharp$ and a function $\gamma_{\mathcal{L}}$, that are used to build the concretisation $\gamma_{\mathcal{V}al}$ of values. In the semantics, a value is either the constant *null* or a location l. The constant *null* can be abstracted by any abstract value $v \in \gamma_{null}$. Locations are non-interpreted, *i.e.*, $\mathcal{L}$ is not a set of concrete addresses in memory, but an artifact of the memory model. Therefore, a concretisation function $\gamma_{\mathcal{L}} : \mathcal{V}al^\sharp \to \mathcal{P}(\mathcal{L})$ would make little sense. The purpose of $\gamma_{\mathcal{L}} : \mathcal{V}al^\sharp \to \mathcal{P}(\mathcal{C}lass \times \mathcal{F} \times \mathcal{IF})$ is to relate in the heap the class of the location and the instrumentation of the fields. As a result $\gamma_{\mathcal{V}al}$ is parametrised by a heap h and is defined as follows:

$$\frac{v^\sharp \in \gamma_{null}}{null \in \gamma^h_{\mathcal{V}al}(v^\sharp)} \qquad \frac{h(l) = (c, o) \qquad \forall f \in c.\ (c, f, o(f)_2) \in \gamma_{\mathcal{L}}(v^\sharp)}{l \in \gamma^h_{\mathcal{V}al}(v^\sharp)}$$

The quantification $\forall f \in c.$ stands for *for all fields f defined in objects of class c*, *i.e.*, f is either defined in c or in a super-class of c. The notation $o(f)_2$ correspond to the second element in the couple $o(f)$: an object o maps a field f to couples (v, i), $o(f)_1$ stands for v, *i.e.*, the value stored in $o.f$, and $o(f)_2$ stand for i, *i.e.*, the flag attached to $o.f$. The rule on the left-hand side simply states that γ_{null} contains all abstract values that can be interpreted as *null*. The other rule states that, given a heap h, a location l belongs to the concretisation of an abstract value $v^\sharp$ only if the object (c, o) at location l has the correct flags attached to its field according to $\gamma_{\mathcal{L}}(v^\sharp)$, *i.e.*, is such that the tuple $(c, f, o(f)_2)$ belongs to the concretisation $\gamma_{\mathcal{L}}(v^\sharp)$, for all the fields f defined in the class c.

The abstraction of environments is defined component-wise, *i.e.,* the abstraction of each variable is non-relational. It is a simple lift of the abstraction of values.

$$\frac{\forall x \in \mathcal{V}ar, e(x) \in \gamma^h_{\mathcal{V}al}(e^\sharp(x))}{e \in \gamma^h_{\mathcal{E}nv}(e^\sharp)}$$

Note that the abstraction of a program defines one abstraction $e^\sharp$ per program point. The abstraction of environments are flow-sensitive but the lift from $\mathcal{E}nv^\sharp$ to $\mathcal{PP} \to \mathcal{E}nv^\sharp$ is not detailed.

The abstraction of the heap is also non-relational and each field is abstracted by a pair of abstract values, the first abstract value is denoted by $h^\sharp(f)_1$ and the second by $h^\sharp(f)_2$.

$$\frac{\begin{array}{l}\forall l \in \mathcal{L}, c \in \mathcal{C}lass, o \in \mathcal{O}bj.\ h(l) = (c, o) \Rightarrow \\ \qquad \forall f \in c.\ (c, f, o(f)_2) \in \gamma_{\mathcal{L}}(h^\sharp(f)_1) \Rightarrow o(f)_1 \in \gamma^h_{\mathcal{V}al}(h^\sharp(f)_2)\end{array}}{h \in \gamma_{\mathcal{H}eap}(h^\sharp)}$$

An abstraction $h^\sharp$ concretises into all heaps h such that for all objects (c, o), if the annotation of the field $o.f$ abides by $h^\sharp(f)_1$—according to $\gamma_{\mathcal{L}}$—then the value of that field must belong to the concretisation of $h^\sharp(f)_2$—according to $\gamma^h_{\mathcal{V}al}$. In other words, given an object, the abstraction of the heap can distinguish fields according to the status of the flag attached to it. For those whose flag is consistent with $h^\sharp(f)_1$, it gives the information $h^\sharp(f)_2$, for the others, it says nothing.

6.1.3 Instantiations of the framework

We intend to verify abstract interpretations obtained by instantiating:

- the domain of annotations $\mathcal{IF}$, the default value at allocation *inull* and the update function *ifield*,
- the abstract domain $\mathcal{V}al^\sharp$ and the concretisations γ_{null} and $\gamma_{\mathcal{L}}$.

Formalisation of part of the BCV. For our core language, the purpose of byte-code verification consists in ensuring that all virtual method calls will succeed. This is the case if for any call instruction $\mathtt{x} := \mathtt{y.c_0.m(a_1, a_2)}$ the class of y is a subclass of $\mathtt{c_0}$. Therefore, to rule out this error, byte-code verification would compute as abstraction for y a class c that is a subclass of $\mathtt{c_0}$.

This analysis does not require any instrumentation of the semantics. Therefore the domain $\mathcal{IF}$ contains only one *dummy* element $\diamond$. The abstract domain $\mathcal{V}al^\sharp$ is defined as $\mathcal{C}lass_\perp$, *i.e.*, an abstract value $v \in \mathcal{V}al^\sharp$ is either a class c which represents either *null* or any object of class $c' \preceq c$, or $\perp$ which represents only *null*.

$$\begin{array}{c}\mathcal{IF} = \{\diamond\} \quad inull = \diamond \quad ifield(i) = \diamond \quad \gamma_{null} = \mathcal{V}al^\sharp \\ \gamma_{\mathcal{L}}(c) = \{(c', f, i) \mid c' \preceq c\}\end{array}$$

Formalisation of a null pointer analysis. Our parametrised concretisation can also model more sophisticated abstract domains similar to the null-pointer type system of Fähnrich and Leino [FL03], presented in Section 2.5.2. The instrumentation of the semantics accounts for the initialisation of fields: at allocation, fields are flaged as *undef*, and the first update in

the field put the flag at *def*. With our semantics, this behaviour is modelled by the following instrumentation:

$$\mathcal{IF} = \{def, undef\} \qquad inull = undef \qquad ifield(i) = def$$

The first part of the abstract domains of the analysis is concerned with the initialisation of fields. It defines an abstract domain $\mathcal{IF}^\sharp$ that contains two values: *Def*, which means *definitively defined*, and *UnDef*, which means *may be defined*, *i.e.*, *UnDef* contains no information.

$$\mathcal{IF}^\sharp = \{Def, UnDef\} \qquad \gamma_{\mathcal{IF}}(Def) = \{def\} \qquad \gamma_{\mathcal{IF}}(UnDef) = \mathcal{IF}$$

Hubert *et al.* [HJP08] have developed an analysis able to infer automatically types that can be checked by type system of Fähnrich and Leino [FL03]. The abstract domain of their analysis differs from the type system in that it requires `this` to be given a more precise abstraction than other variables. Instead of a *raw* type, `this` is abstracted by an explicit mapping $f \in \mathcal{F} \to \{Def, UnDef\}$. In our framework, all the variables are treated in an homogeneous way and doing a special case for `this` would complicate the generic γ. As a result, in our abstraction, all the variables are treated like `this`. This is a generalisation as a raw type $raw(c)$ can be represented by a mapping

$$\lambda f.\ \text{if } f \in c \text{ then } Def \text{ else } UnDef$$

but on the contrary, not all mappings in $\mathcal{F} \to \{Def, UnDef\}$ can be represented by a type $raw(c)$ for some class c.

Another deviation from Fähnrich and Leino or Hubert *et al.,* is that our core language does not contain constructor, only methods and dynamic calls, and any method can be used to initialise an object. The semantics of the `new` statement is simple compared to the JAVA bytecode semantics: `new` just allocates memory but does not call a constructor, if it exists, the initialisation method must be called afterwards. To precisely track down the state of a newly created object of class c, we introduce the type $\hat{c}$ which represents a totally uninitialised object of class exactly c.

We define the abstract domain of values $\mathcal{Val}^\sharp$ as the union of the domains $\mathcal{F} \to \mathcal{IF}^\sharp$, $\widehat{\mathcal{Class}}$, and a third possible domain, $\{MaybeNull, NotNull\}$.

$$\mathcal{Val}^\sharp = \{MaybeNull, NotNull\} \cup \widehat{\mathcal{Class}} \cup (\mathcal{F} \to \mathcal{IF}^\sharp)$$
$$\gamma_{null} = \{MaybeNull\}$$

The type *MaybeNull* represents an arbitrary value and *NotNull* represents a non-null object with all its fields initialised. The type $\hat{c}$ represents an uninitialised object of class (exactly) c and a mapping $F \in \mathcal{F} \to \mathcal{IF}^\sharp$ represents an object such that the initialisation state of a field f is given by $F(f)$.

$$\begin{array}{ll}
\gamma_{\mathcal{L}}(MaybeNull) & = \mathcal{Class} \times \mathcal{F} \times \mathcal{IF} \\
\gamma_{\mathcal{L}}(NotNull) & = \{(c, f, i) \mid f \in c \Rightarrow i = def\} \\
\gamma_{\mathcal{L}}(\hat{c}) & = \{c\} \times \mathcal{F} \times \mathcal{IF} \\
\gamma_{\mathcal{L}}(F) & = \{(c, f, i) \mid f \in c \Rightarrow i \in \gamma_{\mathcal{IF}}(F(f))\}
\end{array}$$

A feature of this analysis is that even if the abstraction of the heap is not flow-sensitive, it can make a distinction between initialised fields and uninitialised fields. This property is obtained as soon as an abstract heap $h^\sharp \in \mathcal{H}eap^\sharp$ is such that $h^\sharp(f)_1 = NotNull$. In fact, the abstractions of the heap calculated by an analyser all belongs to a subset

$$\{NotNull\} \times \mathcal{V}al^\sharp$$

of the possible abstractions in $\mathcal{V}al^\sharp \times \mathcal{V}al^\sharp$. Therefore the results of the analyser will only provide information about the fields of initialised objects, whereas fields of partially initialised objects will be treated as potentially null.

6.2 Simplification of object-oriented VCs

We define a new VC calculus $\mathcal{DVC}_{obj}$, designed to produce verification conditions that belong to a decidable theory $\mathcal{DA}ssn$ based on a fragment of the theory of semantic states and new atoms that use γ_{null} and $\gamma_{\mathcal{L}}$.

6.2.1 A fragment with quantifier elimination

We define the theory $\mathcal{DA}ssn$ of formulae of the form

$$\forall \bar{c} \in \mathcal{C}lass, \bar{f} \in \mathcal{F}, \bar{i} \in \mathcal{IF}, \bar{v} \in \mathcal{V}ar.\ \phi$$

where $\bar{c}, \bar{f}, \bar{i}, \bar{v}$ are vectors of universally quantified variables and ϕ is a quantifier-free propositional formula built over the following atomic propositions

$$p ::= v^\sharp \in \gamma_{null} \mid (c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp) \mid c \preceq c \mid f \in c.$$

Here, $v^\sharp$ is a constant of the abstract domain $\mathcal{V}al^\sharp$; c is either a constant class name or a class variable bound in $\bar{c}$; f is either a constant field or a field variable bound in $\bar{f}$; i is an annotation of the form $ifield^n(i)$ where i is either a constant annotation or an annotation variable bound in $\bar{i}$.

To argue the decidability of $\mathcal{DA}ssn$, we first observe that ground formulae $c \preceq c'$ and $f \in c$ are syntactic properties of programs that are trivially decidable. As a result, the decidability of the ground theory depends only on the abstract interpretation, *i.e.,* the definition of γ_{null} or $\gamma_{\mathcal{L}}$. For any reasonable analysis it is likely that it is decidable whether an abstract element $v^\sharp$ represents *null* ($v^\sharp \in \gamma_{null}$) and whether a triple of constants (c, f, i) is part of $\gamma_{\mathcal{L}}(v^\sharp)$. To deal with quantified formulae, we recall that a theory admits *quantifier elimination* if for any quantified formula there exists an equivalent ground formula. Quantifier elimination provides a way to lift the decidability of ground formulae to the decidability of the quantified fragment. In our case, the only assumption required for decidability is that the domain

$\mathcal{IF}$ is finite. In that case, the previous theory admits quantifier-elimination because each quantified variable ranges over a finite domain, be it $\mathcal{IF}$, $\mathcal{Class}$, $\mathcal{F}$ or $\mathcal{Var}$.

The byte-code verification logic is decidable. There is only a single annotation. Moreover, $v^\sharp \in \gamma_{null}$ always holds and $(c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp)$ reduces to $c \preceq v^\sharp$.

The null-pointer logic is also decidable. In this case, there are only two annotations. Moreover, $v^\sharp \in \gamma_{null}$ is effectively an atom of the theory of equality—it holds if and only if $v^\sharp = \mathit{MaybeNull}$—and $(c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp)$ is a formula combining the theory of equality and existing atoms:

$$\begin{array}{lll}
(c, f, i) \in \gamma_{\mathcal{L}}(\mathit{MaybeNull}) & \text{iff} & \mathit{True} \\
(c, f, i) \in \gamma_{\mathcal{L}}(\mathit{NotNull}) & \text{iff} & f \in c \Rightarrow i = \mathit{def} \\
(c, f, i) \in \gamma_{\mathcal{L}}(\hat{c}') & \text{iff} & c' = c \\
(c, f, i) \in \gamma_{\mathcal{L}}(F) & \text{iff} & f \in c \Rightarrow F(f) = \mathit{Def} \Rightarrow i = \mathit{def}
\end{array}$$

For these restrictions to be of interest we must show that a sound verification condition calculus can be defined in this fragment. This is far from evident, as Remark that the concretisation defined in Section 6.1 uses universal quantification of locations on both sides of implications, therefore if it was used for the verification condition calculus defined in Chapter 4, the VCs would fall into the quantifier-free fragment.

6.2.2 Decidable verification conditions

We define the VC calculus $\mathcal{DVC}_{obj}$ that generates finitely many verification conditions that are provably sufficient to ensure the invariant of an analysis and therefore the absence of run-time errors in a program. The essential property of $\mathcal{DVC}_{obj}$ is that it generates verification conditions that belong to the logic fragment $\mathcal{DAssn}$ identified in Section 6.2.1. The soundness of $\mathcal{DVC}_{obj}$ has been formally proved in Coq [BC04] and is available online [BCJ12].

The verification conditions generated by $\mathcal{DVC}_{obj}$ require the instrumentation to be *heap monotonic w.r.t.* to the abstraction of location:

$$\forall v^\sharp, c, f, i.\ (c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp) \Rightarrow (c, f, \mathit{ifield}(i)) \in \gamma_{\mathcal{L}}(v^\sharp)$$

This property has already been identified as being instrumental for coping with multi-threading [FL03]. It informally states that at every time, the abstraction holds even if the fields are *more defined* than expected, *i.e.*, even if another thread has updated some fields. In a sequential setting, it could be relaxed at the cost of introducing an additional quantification modelling the fact that, for instance, during a method call the instrumentation can be updated an arbitrary number of times.

$\mathcal{DVC}_{obj}$ manipulates terms of the syntax and the functions γ_{null} and $\gamma_{\mathcal{L}}$ seen as predicates. To simplify the notations we use the short-hands

presented in Figure 6.2. The first short-hand, $v_1^\sharp \overset{\bullet}{\Rightarrow} v_2^\sharp$, means that $\gamma_{\mathcal{L}}(v_1^\sharp) \sqsubseteq \gamma_{\mathcal{L}}(v_2^\sharp)$, where $\sqsubseteq$ denotes an inclusion restricted to tuples (c, f, i) in which $f \in c$, and that if $v_1^\sharp$ can denote *null*, so does $v_2^\sharp$. The second short-hand $v_1^\sharp \overset{\bullet}{\wedge} v_2^\sharp \overset{\bullet}{\Rightarrow} v_3^\sharp$ defines a similar relation but with an intersection of states on the left-hand side: $\gamma_{\mathcal{L}}(v_1^\sharp) \sqcap \gamma_{\mathcal{L}}(v_2^\sharp) \sqsubseteq \gamma_{\mathcal{L}}(v_3^\sharp)$, where $\sqcap$ is also restricted to tuples (c, f, i) in which the field f is defined in the class c. The definition of the short-hands translates the inclusion into the theory of semantic states.

$$v_1^\sharp \overset{\bullet}{\Rightarrow} v_2^\sharp \quad \triangleq \quad \bigwedge \begin{array}{l} v_1^\sharp \in \gamma_{null} \Rightarrow v_2^\sharp \in \gamma_{null} \\ \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(v_1^\sharp) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(v_2^\sharp) \end{array}$$

$$v_1^\sharp \overset{\bullet}{\wedge} v_2^\sharp \overset{\bullet}{\Rightarrow} v_3^\sharp \quad \triangleq \quad \bigwedge \begin{array}{l} v_1^\sharp \in \gamma_{null} \wedge v_2^\sharp \in \gamma_{null} \Rightarrow v_3^\sharp \in \gamma_{null} \\ \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(v_1^\sharp) \wedge (c, f, i) \in \gamma_{\mathcal{L}}(v_2^\sharp) \\ \qquad \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(v_3^\sharp) \end{array}$$

Figure 6.2: Short-hands used in the simplified VCs

Given an abstraction $(H, E) \in \mathcal{Heap}^\sharp \times (\mathcal{PP} \to \mathcal{Env}^\sharp)$ of the program, we generate for each program point p a verification condition $VC^\sharp{}_p^{(H,E)}$ for the statement $s \in \mathcal{Stmt}$ such that $\texttt{get_stmt}(p) = s$. For each method signature $m' \in \mathcal{Class} \times \mathcal{Method}$ which overrides a method $m \in \mathcal{Class} \times \mathcal{Method}$ in a subclass, we also generate verification conditions $VC^\sharp(m', m)^{(H,E)}$ modelling the usual variance/co-variance rules for method redefinitions, *i.e.*, the pre-condition of the redefinition should follow from the pre-condition of the initial definition, and the post-condition of the redefinition should imply the post-condition of the initial definition. The comprehensive VCs are given in Figure 6.4 on page 102 and detailed below. In all rules, the terms of the statement on which the VC is produced are capital letters in a True-Type font (*e.g.*, `X`) and the two parts of the abstraction are written in italic capital letters. We do not indicate the sorts of the quantified variables to keep the formulae readable, but all v are variables in $\mathcal{Var}$, c are classes in $\mathcal{Class}$, f are fields in $\mathcal{F}$, i are instrumentations of fields in $\mathcal{IF}$, except in the VC for call instructions, where it is specified $\forall i \in \{0, 1\}$ to avoid repeating the condition.

Skip. The VC for the `skip` instruction is limited to the propagation of the flow-sensitive abstraction of the environment.

$$VC^\sharp(\texttt{skip})_p^{(H,E)} = \forall v.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v)$$

If p is the program point annotated with the `skip` instruction, then for all variables v, the abstraction $E(p)(v)$ should be contained in the abstraction

of the next program-point $E(p^+)(v)$. This condition is also found in all the other VCs, with a restriction on the variable v, *i.e.*, all instructions that modify a variable $\mathtt{X}$ obviously does not have to report the exact same information regarding that variable on the next program point.

Assignments. We produce different VCs for assignments $\mathtt{X} := e$ depending on the expression e. If e is simply `null`, then the VC simply propagates the information on all variables different from $\mathtt{X}$ and checks that the abstract value for $\mathtt{X}$ at the next program point can represent a null value.

$$VC^\sharp(\mathtt{X} := \mathtt{null})_p^{(H,E)} = \begin{cases} \forall v \neq \mathtt{X}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge E(p^+)(\mathtt{X}) \in \gamma_{null} \end{cases}$$

The other VC for assignment deals with instructions of the form $\mathtt{X} := \mathtt{X}'$, and checks that the information on $\mathtt{X}'$ are propagated to the information on $\mathtt{X}$ at the next program point, replacing the condition $E(p^+)(\mathtt{X}) \in \gamma_{null}$ by $E(p)(\mathtt{X}') \overset{\bullet}{\Rightarrow} E(p^+)(\mathtt{X})$.

Method calls. Along the same lines, most of the conditions of the VC for call statements $x := \mathtt{Y.C.M}(\mathtt{V}_0, \mathtt{V}_1)$ simply check that the correct information is propagated. First, the information on all local variables that are not concerned by the call—variables that are neither $\mathtt{X}, \mathtt{Y}, \mathtt{V}_0$ nor $\mathtt{V}_1$—must be propagated to the next program point.

$$\forall v \notin \{\mathtt{X}, \mathtt{Y}, \mathtt{V}_0, \mathtt{V}_1\}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v)$$

Then, the VC must check that the pre-condition of the method called is enforced, *i.e.*, it must check that the information on the argument of the call $\mathtt{Y}$,$\mathtt{V}_0$ and $\mathtt{V}_1$ implies the information on the parameter `this`,p_0 and p_1 at the entry point of the method. We take the entry point of the implementation of the method in the highest possible class $(\mathtt{C}, \mathtt{M})_0$. A different VC checks that all implementations respect the usual variance/co-variance rule for method redefinitions.

$$\forall i, f, c' \preccurlyeq \mathtt{C}.\ (c', f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{Y})) \Rightarrow (c', f, v) \in \gamma_{\mathcal{L}}(E((\mathtt{C}, \mathtt{M})_0)(\mathtt{this}))$$
$$\forall i \in \{0, 1\}.\ E(p)(\mathtt{V}_i) \overset{\bullet}{\Rightarrow} E((\mathtt{C}, \mathtt{M})_0)(p_i)$$

The constraint concerning the parameter `this` is a bit relaxed: we know that the object $\mathtt{Y}$ is not null and at most of class $\mathtt{C}$. A different VC, presented in Figure 6.3 on page 101, is in charge of checking that the lookup never fails. The VC checks that the information on $\mathtt{Y}$ at the call point *up to* $\mathtt{C}$ is propagated to the information on `this` at the entry point.

Finally, the VC checks that the information at the exit point $(\mathtt{C}, \mathtt{M})_\infty$—*i.e.*, the post-condition of the method—is propagated.

$$E((\mathtt{C}, \mathtt{M})_\infty)(\mathtt{res}) \overset{\bullet}{\Rightarrow} E(p^+)(\mathtt{X})$$
$$E((\mathtt{C}, \mathtt{M})_\infty)(\mathtt{this}) \overset{\bullet}{\wedge} E(p)(\mathtt{Y}) \overset{\bullet}{\Rightarrow} E(p^+)(\mathtt{Y})$$
$$\forall i \in \{0, 1\}.\ E((\mathtt{C}, \mathtt{M})_\infty)(\mathtt{p_i}) \overset{\bullet}{\wedge} E(p)(\mathtt{V}_i) \overset{\bullet}{\Rightarrow} E(p^+)(\mathtt{V}_i)$$

Note that even if the semantics specify that the value—*i.e.*, the location—of the variables $\mathtt{Y}$,$\mathtt{V}_0$ and $\mathtt{V}_1$ is not touched by the call, the object they point to may have been modified by the call, *e.g.*, more fields could be initiated. Therefore, the information at the next program point on these variables is actually the intersection of the information at the call point—*i.e.*, $E(p)(\bullet)$—and of the information on the parameters—`this` for $\mathtt{Y}$, $\mathtt{p}_0$ for $\mathtt{V}_0$ and $\mathtt{p}_1$ for $\mathtt{V}_1$—at the exit point of the method $E((\mathtt{C},\mathtt{M})_\infty)(\bullet)$, hence the use of the shorthand $v_1^\sharp \overset{\bullet}{\wedge} v_2^\sharp \overset{\bullet}{\Rightarrow} v_3^\sharp$.

Conditional tests. A program point p annotated with a branching statement $\mathtt{Jnull}(\mathtt{X}, p')$ generates one VC, with conditions related to the two branches. If the information on $\mathtt{X}$ at program point p indicates that the variable can be null, *i.e.*, $E(p)(\mathtt{X}) \in \gamma_{null}$, then the jump may occur, therefore the information on $\mathtt{X}$ at p' must signal that $\mathtt{X}$ may be null, and the information on all other variables must be propagated from p to p'.

$$\begin{array}{l} E(p)(\mathtt{X}) \in \gamma_{null} \Rightarrow E(p')(\mathtt{X}) \in \gamma_{null} \\ \forall v \neq \mathtt{X}.\ E(p)(\mathtt{X}) \in \gamma_{null} \Rightarrow E(p)(v) \overset{\bullet}{\Rightarrow} E(p')(v) \end{array}$$

As soon as the information on $\mathtt{X}$ at p indicates that the variable can be not-null, *i.e.*, if $(c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{X}))$ is true, then some executions may continue to p^+ and the information must be propagated accordingly.

$$\begin{array}{l} \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{X})) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(p^+)(\mathtt{X})) \\ \forall v \neq \mathtt{X}, \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{X})) \Rightarrow E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \end{array}$$

Note that the information that $\mathtt{X}$ may be null at p is not propagated to p^+, we use a constraint a bit more relaxed than a simple $E(p)(\mathtt{X}) \overset{\bullet}{\Rightarrow} E(p^+)(\mathtt{X})$, and can therefore certify *guard-sensitive* analyses.

A simpler VC could account for analyses that do not take the guard into account. It would only need to check that the information is correctly propagated to both possible successor of p

$$\begin{array}{l} \forall v, E(p)(v) \overset{\bullet}{\Rightarrow} E(p')(v) \\ \forall v, E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \end{array}$$

However, such VC could not account for analyses that use the test on $\mathtt{X}$ to infer a more precise information in each branch, and given the nature of the test—*i.e.*, "*is* $\mathtt{X}$ *null ?*"—this would entails a potential loss of precision in the null-pointer analysis.

Object allocation. The VC for the $\mathtt{X} := \mathtt{new\ C}$ statement is straightforward. It only has to check—besides the fact that variables other than $\mathtt{X}$ are unchanged—that the information on $\mathtt{X}$ at the next program point can

account for the fact that all the fields of the object stored in `X` have a blank annotation and have a null value.

$$\forall f \in \mathtt{C}.\ (\mathtt{C}, f, inull) \in \gamma_{\mathcal{L}}(E(p^+)(x))$$
$$\forall f \in \mathtt{C}.\ (\mathtt{C}, f, inull) \in \gamma_{\mathcal{L}}(H(f)_1) \Rightarrow H(f)_2 \in \gamma_{null}$$

Accesses in the heap. The VC for a program point p annotated with an access in the heap $\mathtt{X} := \mathtt{Y.F}$ states that the information on `F` in the flow-insensitive abstraction of the heap, *i.e.*, $H(\mathtt{F})_2$, should be propagated to the information on `X` at the next program point. Nonetheless, recall that the abstraction of the heap may distinguish between the possible annotations of `F`. Therefore, the information from H must be propagated to $E(p^+)(\mathtt{X})$ depending on what the information on `Y` at p can say about the flag on `Y.F`.

$$\begin{array}{l}
\forall c, i. \\
\quad \mathtt{F} \in c \Rightarrow \\
\quad (c, \mathtt{F}, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{Y})) \Rightarrow \\
\quad \Big((c, \mathtt{F}, i) \in \gamma_{\mathcal{L}}(H(\mathtt{F})_1) \Rightarrow H(\mathtt{F})_2 \in \gamma_{null}\Big) \Rightarrow \\
\quad E(p^+)(\mathtt{X}) \in \gamma_{null} \\
\forall c, c', f', i, i'. \\
\quad \mathtt{F} \in c \Rightarrow \\
\quad (c, \mathtt{F}, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{Y})) \Rightarrow \\
\quad \Big((c, \mathtt{F}, i) \in \gamma_{\mathcal{L}}(H(\mathtt{F})_1) \Rightarrow (c', f', i') \in \gamma_{\mathcal{L}}(H(\mathtt{F})_2)\Big) \Rightarrow \\
\quad (c', f', i') \in \gamma_{\mathcal{L}}(E(p^+)(\mathtt{X}))
\end{array}$$

There are two kinds of information to propagate: f may be null, *i.e.*, $H(\mathtt{F})_2 \in \gamma_{null}$, and the set of objects the abstraction of f may correspond to, hence the two conditions. Remark that the parentheses does not allows the use of the shorthand $H(\mathtt{F})_2 \overset{\bullet}{\Rightarrow} E(p^+)(\mathtt{X})$.

Updates in the heap. The VC for a program point p annotated with an update in the heap $\mathtt{X.F} := \mathtt{Y}$ checks that the information on `Y` is propagated in the heap

$$E(p)(\mathtt{Y}) \overset{\bullet}{\Rightarrow} H(\mathtt{F})_2$$

but must also checks that the abstraction accounts for the update on the flag attached to the field. It must be accounted for in the abstraction of the

environment, for all objects in which F is defined, but only for the field F

$$\begin{array}{l}
\forall c, i. \\
\quad \mathtt{F} \in c \Rightarrow \\
\quad (c, \mathtt{F}, i) \in E(p)(x) \Rightarrow \\
\quad (c, \mathtt{F}, ifield(i)) \in E(p^+)(x) \\
\forall i, c, f' \neq \mathtt{F}. \\
\quad \mathtt{F} \in c \Rightarrow \\
\quad (c, f', i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{X})) \Rightarrow \\
\quad (c, f', i) \in \gamma_{\mathcal{L}}(E(p^+)(\mathtt{X}))
\end{array}$$

and it must be accounted for in the heap.

$$\forall c, f', i. \quad (c, \mathtt{F}, i) \in \gamma_{\mathcal{L}}(E(p)(x)) \Rightarrow (c, \mathtt{F}, \mathit{ifield}(i)) \in \gamma_{\mathcal{L}}(H(f')_2)$$

Absence of errors. For each statement $s \in \mathcal{S}tmt$ at program point p which can potentially be responsible for an error $e \in \mathcal{E}rr$ we generate an abstract verification condition $Chk^\sharp(s)_p^{(H,E)}$ ruling out this error. Figure 6.3 details the VCs for the statement that can generate errors: for accesses and updates in the heap, VCs that rule out null pointer dereferencing, and for method calls, a VC that also rules out potential failures of the lookup algorithm.

$$\begin{array}{ll}
Chk^\sharp(\mathtt{x} := \mathtt{y.f})_p^{(H,E)} & = \neg(E(p)(\mathtt{y}) \in \gamma_{null}) \\
Chk^\sharp(\mathtt{x.f} := \mathtt{y})_p^{(H,E)} & = \neg(E(p)(\mathtt{x}) \in \gamma_{null}) \\
Chk^\sharp(\mathtt{x} := \mathtt{y.c.m}(\mathtt{v_0}, \mathtt{v_1}))_p^{(H,E)} & = \left\{ \begin{array}{l} \neg(E(p)(\mathtt{y}) \in \gamma_{null}) \\ \wedge \forall c', f, i.\ (c', f, i) \in E(p)(\mathtt{y}) \Rightarrow c' \preceq \mathtt{c} \end{array} \right.
\end{array}$$

Figure 6.3: Verification conditions proving the absence of errors

Definition 6.2.1. We write $\mathcal{DVC}_{obj}$ the VC calculus defined by the rules presented in Figures 6.4 and 6.3:

Let P be a well-formed program encoded as a flowchart and (H, E) be the untrusted result of an analysis such that the instrumentation is monotonic.

- $\forall p \in \mathcal{PP}.\ VC^\sharp{}_p^{(H,E)}(\mathtt{get_stmt}(p)) \in \mathcal{DVC}\,(P, (H, E))$.
- $\forall m, m'$. if m' is a redefinition of m, $VC^\sharp(m', m)^{(H,E)} \in \mathcal{DVC}\,(P, (H, E))$.
- $\forall p \in \mathcal{PP}$, if $s = \mathtt{get_stmt}(p)$ can potentially be responsible for an error, $Chk^\sharp(s)_p^{(H,E)} \in \mathcal{DVC}\,(P, (H, E))$.

$$
\begin{array}{lcl}
VC^\sharp(\texttt{skip})_p^{(H,E)} & = & \forall v.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\
VC^\sharp(\texttt{X} := \texttt{null})_p^{(H,E)} & = & \left\{ \begin{array}{l} \forall v \neq \texttt{X}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge E(p^+)(\texttt{X}) \in \gamma_{null} \end{array} \right. \\
VC^\sharp(\texttt{X} := \texttt{X}')_p^{(H,E)} & = & \left\{ \begin{array}{l} \forall v \neq \texttt{X}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge E(p)(\texttt{X}') \overset{\bullet}{\Rightarrow} E(p^+)(\texttt{X}) \end{array} \right. \\
VC^\sharp(x := \texttt{Y.C.M}(\texttt{V}_0, \texttt{V}_1))_p^{(H,E)} & = & \left\{ \begin{array}{l} \forall v \notin \{\texttt{X}, \texttt{Y}, \texttt{V}_0, \texttt{V}_1\}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge \forall i, f, c' \preccurlyeq \texttt{C}.\ (c', f, i) \in \gamma_{\mathcal{L}}(E(p)(\texttt{Y})) \Rightarrow (c', f, v) \in \gamma_{\mathcal{L}}(E((\texttt{C},\texttt{M})_0)(\texttt{this})) \\ \wedge \forall i \in \{0,1\}.\ E(p)(\texttt{V}_i) \overset{\bullet}{\Rightarrow} E((\texttt{C},\texttt{M})_0)(p_i) \\ \wedge E((\texttt{C},\texttt{M})_\infty)(\texttt{res}) \overset{\bullet}{\Rightarrow} E(p^+)(\texttt{X}) \\ \wedge E((\texttt{C},\texttt{M})_\infty)(\texttt{this}) \overset{\bullet}{\wedge} E(p)(\texttt{Y}) \overset{\bullet}{\Rightarrow} E(p^+)(\texttt{Y}) \\ \wedge \forall i \in \{0,1\}.\ E((\texttt{C},\texttt{M})_\infty)(\texttt{p}_\texttt{i}) \overset{\bullet}{\wedge} E(p)(\texttt{V}_i) \overset{\bullet}{\Rightarrow} E(p^+)(\texttt{V}_i) \end{array} \right. \\
VC^\sharp(\texttt{Jnull}(\texttt{X}, p'))_p^{(H,E)} & = & \left\{ \begin{array}{l} E(p)(\texttt{X}) \in \gamma_{null} \Rightarrow E(p')(\texttt{X}) \in \gamma_{null} \\ \wedge \forall v \neq \texttt{X}.\ E(p)(\texttt{X}) \in \gamma_{null} \Rightarrow E(p)(v) \overset{\bullet}{\Rightarrow} E(p')(v) \\ \wedge \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\texttt{X})) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(p^+)(\texttt{X})) \\ \wedge \forall v \neq \texttt{X}, \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\texttt{X})) \Rightarrow E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \end{array} \right. \\
VC^\sharp(\texttt{X} := \texttt{new C})_p^{(H,E)} & = & \left\{ \begin{array}{l} \forall v \neq \texttt{X}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge \forall f \in \texttt{C}.\ (\texttt{C}, f, inull) \in \gamma_{\mathcal{L}}(E(p^+)(x)) \\ \wedge \forall f \in \texttt{C}.\ (\texttt{C}, f, inull) \in \gamma_{\mathcal{L}}(H(f)_1) \Rightarrow H(f)_2 \in \gamma_{null} \end{array} \right. \\
VC^\sharp(\texttt{X} := \texttt{Y.F})_p^{(H,E)} & = & \left\{ \begin{array}{l} \forall v \neq \texttt{X}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge \forall c, i. \\ \quad \texttt{F} \in c \Rightarrow \\ \quad (c, \texttt{F}, i) \in \gamma_{\mathcal{L}}(E(p)(\texttt{Y})) \Rightarrow \\ \quad ((c, \texttt{F}, i) \in \gamma_{\mathcal{L}}(H(\texttt{F})_1) \Rightarrow H(\texttt{F})_2 \in \gamma_{null}) \Rightarrow \\ \quad E(p^+)(\texttt{X}) \in \gamma_{null} \\ \wedge \forall c, c', f', i, i'. \\ \quad \texttt{F} \in c \Rightarrow \\ \quad (c, \texttt{F}, i) \in \gamma_{\mathcal{L}}(E(p)(\texttt{Y})) \Rightarrow \\ \quad ((c, \texttt{F}, i) \in \gamma_{\mathcal{L}}(H(\texttt{F})_1) \Rightarrow (c', f', i') \in \gamma_{\mathcal{L}}(H(\texttt{F})_2)) \Rightarrow \\ \quad (c', f', i') \in \gamma_{\mathcal{L}}(E(p^+)(\texttt{X})) \end{array} \right. \\
VC^\sharp(\texttt{X.F} := \texttt{Y})_p^{(H,E)} & = & \left\{ \begin{array}{l} \forall v \neq \texttt{X}.\ E(p)(v) \overset{\bullet}{\Rightarrow} E(p^+)(v) \\ \wedge \forall c, i. \\ \quad \texttt{F} \in c \Rightarrow \\ \quad (c, \texttt{F}, i) \in E(p)(x) \Rightarrow \\ \quad (c, \texttt{F}, ifield(i)) \in E(p^+)(x) \\ \wedge \forall i, c, f' \neq \texttt{F}. \\ \quad \texttt{F} \in c \Rightarrow \\ \quad (c, f', i) \in \gamma_{\mathcal{L}}(E(p)(\texttt{X})) \Rightarrow \\ \quad (c, f', i) \in \gamma_{\mathcal{L}}(E(p^+)(\texttt{X})) \\ \wedge E(p)(\texttt{Y}) \overset{\bullet}{\Rightarrow} H(\texttt{F})_2 \\ \wedge \forall c, f', i. \\ \quad (c, \texttt{F}, i) \in \gamma_{\mathcal{L}}(E(p)(x)) \Rightarrow (c, \texttt{F}, ifield(i)) \in \gamma_{\mathcal{L}}(H(f')_2) \end{array} \right. \\
\end{array}
$$

$$
VC^\sharp(m', m)^{(H,E)} = \left\{ \begin{array}{l} E(m'_\infty)(\texttt{res}) \overset{\bullet}{\Rightarrow} E(m_\infty)(\texttt{res}) \\ \wedge \forall c \preccurlyeq class(m'), f, i.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(m_0)(\texttt{this})) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(m'_0)(\texttt{this})) \\ \wedge \forall i \in \{0,1\}.\ E(m_0)(p_i) \overset{\bullet}{\Rightarrow} E(m'_0)(p_i) \\ \wedge \forall v \notin \{\texttt{this}, \texttt{p}_0, \texttt{p}_1\}.\ E(m_0)(v) \in \gamma_{null} \end{array} \right.
$$

Figure 6.4: Verification conditions proving the soundness of the result of an analysis

Theorem 6.2.1. *The VC calculus $\mathcal{DVC}_{obj}$ is sound:*

Let P be a program encoded as a flowchart and (H, E) be the untrusted result of an analysis such that the instrumentation is monotonic. If all the verification conditions are valid

$$\forall \Phi \in \mathcal{DVC}_{obj}(P, (H, E)), \Phi \text{ is valid}$$

the absence of run-time error is guaranteed by (H, E).

$$\forall s \in \mathcal{State}, e \in \mathcal{Err}, s \in \mathcal{Reach} \Rightarrow s \not\rightsquigarrow e$$

Proof. This theorem is proved correct in the Coq development [BCJ12] in two stages. First Lemma 6.2.2 establish that the VC described in Figure 6.4 ensure the soundness of the analysis result, then Lemma 6.2.3 establish that the VC described in Figure 6.3 ensure the absence of error in P. Section 6.3.2 will explain how this proof could be related to the implementation in Why3.

Lemma 6.2.2. *Let P be a program and (H, E) be the untrusted result of an analysis such that the instrumentation is monotonic. If the VCs hold for all the statements*

$$\forall p \in \mathcal{PP}, s \in \mathcal{Stmt}.\ \texttt{get_stmt}(p) = s \Rightarrow VC^{\sharp}{}_{p}^{(H,E)}(s)$$

and the VCs hold for method redefinitions

$$\forall m, m'.\ override(m', m) \Rightarrow VC^{\sharp}(m', m)^{(H,E)}$$

the analysis result is sound.

$$\mathcal{Reach} \subseteq \gamma(H, E)$$

Lemma 6.2.3. *Let P be a program and (H, E) be a sound analysis result ($\mathcal{Reach} \subseteq \gamma(H, E)$). If the VCs hold for all the statements potentially responsible for an error in P*

$$\forall p \in \mathcal{PP}, s \in \mathcal{Stmt}, \texttt{get_stmt}(p) = s \Rightarrow Chk^{\sharp}{}_{p}^{(H,E)}(s)$$

then the absence of errors is guaranteed by the static analysis result.

$$\forall s \in \mathcal{State}, e \in \mathcal{Err}, s \in \mathcal{Reach} \Rightarrow s \not\rightsquigarrow e \qquad \square$$

6.3 Second experiment: result certification of object-oriented analyses

The present section reports a second experiment illustrating our approach on the two object oriented analyses formalised in Section 6.1.2.

6.3.1 Object-oriented language VCgen

We used Why3 exceptions to represent error states. They are raised by the interpreter when semantic conditions are not met, therefore, semantics conditions leading to error states are integrated in the interpreter's control flow. Exceptions leads to exceptional post-conditions. If the analysis is supposed to eliminate the corresponding error, the exceptional post-condition is set to `false`, and will lead to verification conditions ensuring the absence of errors. If the analysis can't prove the absence of a particular error, the corresponding exceptional post-condition is set to `true` and won't lead to any verification condition.

Branches of the interpreter that should not be reachable are cut using the Why3 keyword *absurd*. In the interpreter, it is used to eliminate a branch corresponding to a dangling pointer. The generated proof obligation will be provable using the assumption that the current state is *well-formed*, according to Definition 2.4.1 and Proposition 2.4.1.

Semantic auxiliary functions can be either implemented or axiomatised, depending on the tool used to discharge the verification condition and its ability to take into account the constructions used in the implementation to guide its proof search.

6.3.2 Generation of verification condition revisited

To implement the VC calculus $\mathcal{DVC}_{obj}$ described in Section 6.2.2, we use the same methodology as with the VC calculus $\mathcal{VC}_{num}$ in Section 5.2.2. We implement the parametrised concretisation and abstract domain described in Section 6.1.2 in the IVL and specify $\mathcal{DVC}_{obj}$ using generalised lemma, universally quantified over program points, class, and methods, as shown in Listing 6.1.

```
type pp
     ...
type abs_val
predicate γ_null abs_val
predicate γ_Val abs_val (class,field,ifield)
type abs_env = map var abs_val
predicate γ_Env (e♯:abs_env) (h:heap) (e:env) = ...
         ...
function (H,E) : abstract_state

lemma generalised_DVC_stmt_i :
  ∀ p:pp. get_stmt p = stmt_i  → VC♯(stmt_i)_p^{H,E}
lemma generalised_DVC_override :
  ∀ c c':class, m m':method.
     override (c',m') (c,m)  →  VC♯((c',m'),(c,m))
```

Listing 6.1: Specification of $\mathcal{DVC}$

With these generalised lemmas, the Theorem 6.2.1 can[1] be proved, *i.e.*, the soundness of $\mathcal{DVC}_{obj}$ *w-r-t* the VC calculus $\mathcal{VC}_{obj}$ described in Chapter 4, once and for all, in the same way the soundness of $\mathcal{VC}_{obj}$ can be proved *w-r-t* the interpreter. WHY3 generates proof-obligations for the specification of $\mathcal{VC}_{obj}$, that can be discharged in COQ using the specification of $\mathcal{DVC}_{obj}$.

To certify the result of an analyser, the program is specified in WHY3 as a flowchart, the $\mathcal{Val}^\sharp$ domain, the γ_{null} and $\gamma_{\mathcal{L}}$ functions are implemented, and the result of the analyser is specified in WHY3. The proof-obligations generated by the WHY3 WP calculus on the generalised lemmas specifying $\mathcal{DVC}_{obj}$ constitute the VC to be discharged, and have to be proved automatically by ATPs.

6.3.3 Results

All experiments [BCJ12] were done on a laptop—the same as for the previous experiments—running Linux with 4GB memory and Inter Core 2 Duo CPU at 2.93GHz. We used Why3 0.71 version and a combination of SMT solvers (Z3 [dMB08c] 2.2, Alt-ergo [CCK06] 0.94) and a TPTP solver (E [Sch02] 1.4). We tested the ability of the automatic provers to discharge the VCs produced by $\mathcal{DVC}_{obj}$ on short programs annotated with abstraction in the byte-code verification logic and in the null-pointer logic. We put emphasis during the tests on method calls and inheritance. As previously, all verification conditions were *eventually* discharged automatically.

More precisely, $\mathcal{DVC}_{obj}$ was specified using a number of nested predicates and functions, used for example to describe the translation of the abstract domain to assertions, or factorise parts of the VCs, and some inlining and splitting was required. However, contrary to the experiment on numerical analyses, the transformations featured in Why3 were not sufficient to obtain the behaviour we desired, and we needed to project the VCs on program points—albeit in a systematic way—using auxiliary *projected* lemmas, as explained in Section 5.2.2.

The VCs functions defining $\mathcal{DVC}_{obj}$, described in Section 6.2.2, are conjunction of assertions. Schematically, one lemma was produced per assertion and per relevant program point, then one lemma per program point stated that "it validated its VC", and finally a lemma stated that all program points validated their VCs (cf Listing 6.1 on the previous page). In this experiment the auxiliary lemmas were generated systematically and not only when needed, and the only Why3 transformation applied were some further inlining on a few lemmas.

The proof-obligations were discharged in less than 5s by one prover or another—most in less than 1s even. As in the previous experiment, the performances of the ATPs are hard to predict and to explain. Again, no

[1] This step has not been implemented (yet): $\mathcal{DVC}obj$ is specified in WHY3 but the proof of its soundness has been done separately, albeit in COQ.

ATP could prove all lemmas and a combination of tools was necessary to achieve full automation. In our experiments, only Alt-Ergo and E were necessary to discharge all goals. Again, Z3 either discharged a goal very quickly or reached timeout whereas the other tools exhibited a larger range of behaviours. As they belong to the same family of provers, the goals proved by Z3 were consistently more often discharged by Alt-Ergo than by E, but neither was able to discharge, alone, all of them.

Note that the final lemma, stating that all program point validated their VCs, could not be discharged using an ATP—even if each auxiliary projected lemma could be—but was proved in Coq instead. However, the same one-line proof, chaining some basic tactics, was reused for all programs and all analyses. Therefore, overall full automation was attained. Similarly, some predicates related to the class hierarchy needed to be tabulated, as needed to be the lemmas stating that the verification conditions hold for method redefinitions. This tabulation took the form of a number of auxiliary lemmas generated automatically, all proved by the ATPS, and one final lemma quantified over the possible redefinitions, which needed a one-line proof in Coq—again, the same one-line Coq proof for all programs.

6.4 Conclusion

For some analyses, a straightforward VC calculus will produce formulae which makes extensive use of quantifiers. This will make it practically impossible to use ATPs to discharge the VCs. We present, for a family of object oriented analyses, a method for defining a more efficient VC calculus, more amenable to ATPs, while still being provably sound. The family of object oriented analyses is defined using a parametrisation of the operational semantics, and parametrised abstract domains and concretisations. This allows our approach to retain generality without sacrificing automation of the result certification.

Our approach has been validated through an implementation with the Why3 tool which is capable of verifying analysis results in a few seconds using *off-the-shelf* solvers. For the experiments, we have developed result certifiers for different static analyses:

- a byte code verifier (BCV) for Java: BCV is a fairly straightforward data flow analysis but requires a semantic model of the heap for its correctness that leads to complex VCs;
- a null-pointer analysis: a more complex analysis than BCV, leading to quantifier-rich VCs where our technique for quantifier elimination is a necessary ingredient for the result certifier to work.

6.4.1 Further work

Presently, VCs are discharged consistently by a combination off-the-shelf TPTP provers and SMT solvers, but their behaviours are unpredictable. It would be desirable to fine-tune the settings of the ATPs to facilitate and accelerate enumeration on finite domains. It is often said that choosing the ideal settings for a particular application can have a tremendous impact on the performances, and even the results, of the ATPs. In our approach, all VCs were discharged by ATPs launched with the default settings used by the Why3 framework, but the time spent on some VCs was unpredictable. A comprehensive study of the effect of different choice of settings on performances for the result certification of a particular analysis may alleviate concern on scalability.

Relying on ATPs for each and every part of the proofs can be seen as a *brute force* approach. A solution to help the ATPs to discharge problematic VCs would be to identify problematic construction or patterns in formulae and isolate them through lemmas proved once by other means, or rely on some auxiliary lemmas generated systematically and proved by generic tactics in a proof-assistant. Ideally, additional decision procedure for some parts of the *semantic state theory* could be developed and integrated into SMT solvers.

We also intend to experiment the framework on other type of analyses. For numerical analyses (interval and polyhedral-based), preliminary experiments indicate that these analyses can be verified using more quantifier-full VCs without putting the SMT solvers into difficulty. Other numerical analyses could be explored to validate this claim, such as *linear congruence equalities* domain [Gra91]. More challenging is the verification of control flow and points-to analyses. These analyses are close to the object oriented analyses presented here but rely on annotations of objects to keep track of allocation points, rather than annotations of fields to keep track of the initialisation status of an object. If $\mathcal{AP}$ is the set of possible allocation points—possibly a subset of the set of program points $\mathcal{PP}$ or a combination of $\mathcal{PP}$ and control-flow information—the abstract domain of value can be define as the power-set of allocation points $\mathcal{Val}^\sharp = \mathcal{P}(\mathcal{AP})$, and the domain of object instrumentation IO—an addition to the parametrisation presented in Section 6.1—can be defined as equal to the set of allocation points $\mathit{IO} = \mathcal{AP}$. The concretisation $\gamma^h_{\mathcal{Val}}(v^\sharp)$ of a value is the set of objects possibly allocated at a point included in $v^\sharp$.

$$\frac{h(l) = (c, o, i) \qquad i \in v^\sharp}{l \in \gamma^h_{\mathcal{Val}}(v^\sharp)}$$

The abstraction of the heap is a mapping between fields and couples of abstract values in $\mathcal{Val}^\sharp$, *e.g.*, $(p_1, p_2) \in h^\sharp(f)$ corresponds to the information: if an object has been allocated at p_1, then the field f points to an

object allocated at p_2. Therefore the concretisation of an abstraction of heaps can be built upon the concretisation $\gamma^h_{\mathcal{V}al}$ of abstractions of objects. The parametrised concretisation we defined in Section 6.1 can be adapted to account for such concretisation of values: γ_{null} and $\gamma_{\mathcal{L}}$ can be defined differently so as to be used to define the concretisation $\gamma^h_{\mathcal{V}al}$, which in turn is used to define $\gamma_{\mathcal{H}eap}$ and $\gamma_{\mathcal{E}nv}$. The VC calculus also needs to be adapted, but the similarities between the structures of the concretisations lead us to suspect the required modification to be marginal.

More generally, a more systematic approach to the definition of dedicated VC calculus would be desirable. Defining a VC calculus dedicated to an analysis or a family of analyses, such as $\mathcal{DVC}_{obj}$, can be seen as a middle-ground between a full proof of the analysis and the general VC calculus $\mathcal{VC}_{obj}$ defined in Chapter 4, which relies on a direct translation of abstract domains to assertions. By providing a framework of parametrised analyses we attain some generality, but finding a parametrisation of the concretisation—and the corresponding VC calculus—appropriate for more analyses, or even defining a systematic approach to the definition of such a parametrisation, that result in a VC that can be discharged by ATPs, remains an open issue. The challenge lies not in defining a sound VC calculus, but in defining a sound VC calculus that belongs to a fragment of the logic that offer strong guaranties that the VC can be discharged consistently by ATPs.

Another challenge would be the result certification of an analyser based on a combination of analyses. In particular, understanding how the reduced product may be taken into account in a VC calculus belonging to the quantifier-free fragment, and how to profit from the modularity it brings to the definition of new analyses, would be a great step towards result certification of pragmatic static analysis.

To conclude, our experiments demonstrated the potential of the approach, but more tests would be necessary to assess its scalability. Extensive tests would require full automation of the translation to Why3 of both programs and analysis result, and finding a significant benchmark would ask for a less restrictive programming language or even a full modern programming language (*e.g.*, JavaCard), hence would require additional VCs. Overall, a much larger implementation effort that could not be justified until the ability of automated theorem provers to consistently discharge the verification condition was demonstrated.

Chapter 7

Satisfiability Modulo Theory background

Automatic Theorem Provers (ATPs) constitute the biggest part of the Trusted Computing Base (TCB) of our static analysis result certification approach. Our tests emphasise the importance of using different tools to achieve satisfactory automation, *i.e.*, for discharging all Verification Conditions (VCs) automatically, and sometimes even for discharging all parts of a single VC. Moreover, satisfiability is an NP complete problem and ATPs rely on aggressive optimisations and heuristics to be efficient. Even though the decision procedures are sound, their implementations can make all kind of errors, which may result in the validation of *non-valid* VCs. Static analyses are be used to prove properties related to security—*e.g.*, the absence of direct accesses in the memory, the correct initialisation of classes and objects. If they make an error and judge a program *safe*, whereas it is not, at least one of the VCs will not be valid. Therefore if Automated Theorem Provers make an error and judge a VC valid whereas it is not, it may result in the certification of an unsafe program. Crafting an attack around such errors may be difficult, but is possible.

To remove ATPs from the Trusted Computing Base, Chapter 8 presents a result certification scheme for a family of ATPs: Satisfiability Modulo Theory (SMT) solvers. The present chapter recalls the necessary background in Satisfiability Modulo Theory necessary for the exposition of our scheme. First, a brief presentation of the *lazy* SMT approach is given in Section 7.1. Then, Section 7.2 discusses proof checking approaches, and benefits of the reconstruction of SMT solvers' results in proof-assistants.

7.1 Lazy SMT approach

This section gives an overview of the essential concepts used in state-of-the-art SMT solvers. It presents SMT solving in three layers, and our proof

format follows closely this layered presentation. Note that we focus on formulae that must be proved unsatisfiable. The certification of satisfiability results is in general significantly simpler, as it amounts to the verification that a given valuation of the variables is a model of the formula. Furthermore, we have reduced the result certification of static analyses to the validity of VCs, which is equivalent to the unsatisfiability of their negations.

We take as running example the quantifier free many sorted formula presented in Figure 7.1, that mixes specifically the theories of equality and Uninterpreted Functions (UF) and Linear Real Arithmetic (LRA).

$$f(f(x)-f(y)) \neq f(z) \ \wedge \ x \leq y \ \wedge \ ((y+z \leq x \wedge z \geq 0) \vee (y-z \leq x \wedge z < 0))$$

Figure 7.1: Running example of a many sorted formula

For UF, a literal is an equality between many sorted ground terms and a formula is a conjunction of positive and negative literals. The axioms of this theory are reflexivity, symmetry and transitivity, and the congruence axiom for functions, as specified in Section 4.2.1 and summarised in Figure 7.2.

$$\forall x.\ x = x$$
$$\forall x, y.\ y = x \implies x = y$$
$$\forall x, y, z.\ x = y \wedge y = z \implies x = z$$
$$\forall x, y.\ x = y \Rightarrow f(x) = f(y)$$

Figure 7.2: Summary of the UF theory

For LRA, a literal is a linear constraint where $(c_i)_{i=0..n} \in \mathbb{Q}$ is a sequence of rational coefficients, $(x_i)_{i=1..n}$ is a sequence of real unknowns and $\bowtie$ is a binary relation.

$$c_0 + c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie 0 \qquad \bowtie \in \{=, >, \geq\}$$

Following Simplify [DNS05], our presentation consider that disequality is managed on the UF side. Therefore, a LRA formula is a conjunction of positive literals.

7.1.1 SAT decision procedure modulo theory

An *eager* approach to the satisfiability of many sorted formulae is to encode the whole formulae into propositional logic [Ack54, ZG03, FG02], reducing the problem to propositional satisfiability. To be practical, such an encoding must avoid the potential exponential blow-up during the encoding, even though the propositional satisfiability problem is already NP. Eager

encoding for the UF theory are well-studied, but to solve the satisfiability of formulae combining other theories, such as LRA, and to avoid the growth in propositional formulae, modern SMT solver rely on a different, *lazy*, approach [dMR02, ABC+02, FJOS03], that uses a (propositional) SAT decision procedure to solve the propositional component of the problem and a *theory engine* to reason about the combination of theories.

The lazy SMT solver approach abstracts each many sorted atom of the input formula by a distinct propositional variable, uses a SAT solver to find a propositional model of the propositional abstraction, and then checks that model against the theory using the theory engine. Models that are incompatible with the theories are discarded by adding a proper lemma to the original formula. This process is repeated until all possible propositional models have been explored.

Figure 7.3 presents the result of the propositional abstraction on our running example presented in Figure 7.1. As it can be seen on this example, the abstraction is not completely oblivious to the theories, and can observe that $z < 0$ is the negation of $z \geq 0$. A theory aware abstraction allows to share terms, *i.e.*, it can reuse the same propositional variables—$z < 0$ is abstracted by $\neg D$ rather than by a new variable. This sharing of terms result in a shorter proof search, but even a naive abstraction would suffice.

a) Propositional abstraction of the initial formula:

$$A \wedge B \wedge ((C \wedge D) \vee (E \wedge \neg D)) \tag{7.1}$$

b) Mapping from propositional variables to many sorted atoms:

A	B	C	D	E
$f(f(x) - f(y)) \neq f(z)$	$x \leq y$	$y + z \leq x$	$z \geq 0$	$y - z \leq x$

(7.2)

Figure 7.3: Setup of a lazy SMT solving of the running example formula presented in Figure 7.1

The valuation σ_1 defined below (7.3) is a possible model of the propositional abstraction (7.1). Using the mapping (7.2) from propositional variables to multi-theory atom, the SMT solver then translates back the model σ_1 into a multi-theory conjunction (7.4).

$$\sigma_1 \mathrel{\hat{=}} A\colon \mathit{true}, B\colon \mathit{true}, C\colon \mathit{true}, D\colon \mathit{true}, E\colon \mathit{false} \tag{7.3}$$

$$(f(f(x)-f(y)) \neq f(z)) \wedge (x \leq y) \wedge (y+z \leq x) \wedge (z \geq 0) \wedge \neg(y-z \leq x) \tag{7.4}$$

Consider the many-sorted FOL formula (7.4) obtained from σ_1. If (7.4) is satisfiable, then the propositional model σ_1 can be translated into a many-sorted model of the initial formula, presented in Figure 7.1. Conversely, if

(7.4) is unsatisfiable, σ_1 can not be translated in a model of the initial formula, and another propositional model must be tested. To obtain a *new* model from the SAT engine, the model σ_1 must first be discarded. To do so, the clause (7.5), called a *conflict clause*, is added to the context of the SAT engine.

$$\neg(A \wedge B \wedge C \wedge D \wedge \neg E) \tag{7.5}$$

This process—asking for a propositional model, testing it with the theory engine, discarding it by adding a conflict clause to the context of the SAT engine—is repeated until all the propositional models are proved unsatisfiable by the theory, *i.e.*, once the conjunction of the propositional abstraction and all the conflict clauses is unsatisfiable. The SMT solver then concludes to the unsatisfiability of the initial many-sorted formula.

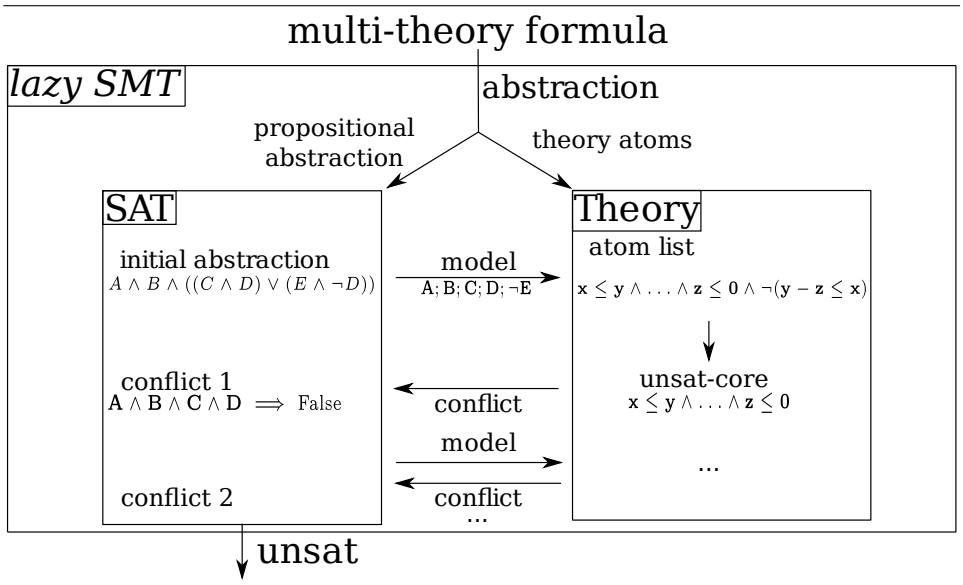


Figure 7.4: Lazy SMT approach applied to the running example presented in Figure 7.1

To accelerate the search, the theory engine usually returns *unsat cores*, *i.e.*, minimal subsets of a propositional model still unsatisfiable for the theories. These unsat cores lead to stronger conflict clauses, as illustrated by the Figure 7.4 with the following clause:

$$\neg(A \wedge B \wedge C \wedge D) \tag{7.6}$$

Whereas the conflict clause (7.5) only discarded the model σ_1 (7.3), this new conflict also discarded all models that could lead to the same unsat core, such as the valuation σ_2 mapping E to *true*.

$$\sigma_2 \mathrel{\hat{=}} A\colon\ true, B\colon\ true, C\colon\ true, D\colon\ true, E\colon\ true$$

The lazy SMT approach as it was presented here requires a SAT engine and a theory engine such that:

- the SAT engine implements an *incremental* search for models, *i.e.*, the SMT proof search needs to add clauses to the context of the SAT engine's context,
- the theory engine implements a multi-theory decision procedure for the satisfiability of conjunctions that can output unsat cores.

7.1.2 Multi-theory decision procedure for conjunctions

In the previous steps, the theory solvers have been fed with conjunctions of many sorted literals. We now explain the Nelson-Oppen (NO) algorithm that is a sound and complete Decision Procedure (DP) for combining infinitely stable theories with disjoint signatures [NO79]. Figure 7.5 presents the deduction steps of this procedure on the theory lemma (7.4). We start from the formula at the top of Figure 7.5 and first apply a *purification* step that introduces sufficiently many intermediate variables to flatten each terms and dispatch *pure* formulae to each theory. Then, each theory exchanges new equalities with the others, until a contradiction is found.

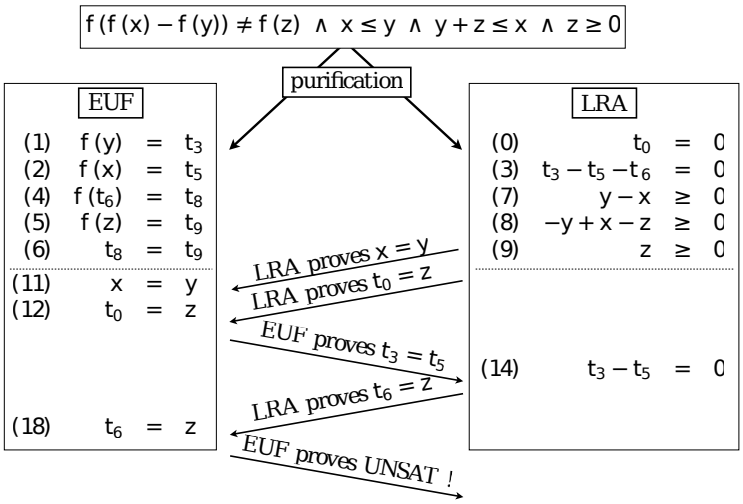


Figure 7.5: Example of Nelson-Oppen equality exchange

To deduce an unsat core from an Nelson-Oppen equality exchange, on can keep track of the hypothesis that are actually used during the exchange. This can not guaranty the minimality of the unsat core, and even achieving non-redundancy can raise significant algorithmic challenges [NO07], depending on the decision procedures used for the theories. However, the minimality of unsat cores only accelerates the SMT search, and is not necessary to

deduce the unsatisfiability. To be efficiently integrated inside the Nelson-Oppen algorithm, the cooperating decision procedures must be incremental, *i.e.*, each theory DP must be able to assert, during a proof search, the equality of two variables.

7.1.3 Optimisations of the lazy SMT approach

If the use of unsat cores can be easily described in the lazy proof search as it was presented in Section 7.1.1, modern SMT solver implement numerous other optimisations. First of all, the theory engine implements a richer interface, allowing a closer integration of the theory engine into a DPLL proof search [DP60, DLL62]—the standard algorithm for propositional satisfiability. The resulting decision procedure is referred to as the DPLL(T) algorithm [GHN+04, NOT06]. This better integration allows the SMT solvers to check partial models incrementally against the theory in order to detect conflicts earlier, and the multi-theory engine may discover propagation lemmas [ACG00, ABC+02, GHN+04], *i.e.*, theory literals that are consequence of partial models. On the running example, this would mean understanding that the partial model

$$A\colon\ \mathit{true}, B\colon\ \mathit{true}, C\colon\ \mathit{true}, D\colon\ \mathit{true}$$

leads to a contradiction, and adding the conflict clause (7.6) *before* attributing a value to E.

The Nelson-Oppen decision procedure requires theory decision procedures to provide, on satisfiable conjunctions, the list of all entailed equalities between variables, which may be particularly costly for some theories. Furthermore, formulae belonging to non-convex theories, *e.g.*, the Linear Integer Arithmetic (LIA) decision procedure may need to propagate *disjunctions* of equalities between variables, instead of conjunctions. For example, the following system of constraints

$$a \leq b \leq c \wedge c = a + 1$$

entails no equalities in LRA. However, in LIA, the disjunction $a = b \vee b = c$ can be deduced, but neither $a = b$ nor $b = c$ are consequences on their own. To integrate these theories, the Nelson-Oppen algorithm can be enriched with case-split and backtrack features. Or, rather than modifying the Nelson-Oppen algorithm, the cooperation between theories may be passed over to the SAT engine, by means of delayed theory combination [BBC+05, BCF+06] or model based theory combination [dMB08a].

7.2 Checking SMT proofs

Even if all applications based on Satisfiability Modulo Theory solvers did not require more than a verdict of satisfiability, SMT solving algorithms involve

different, complex tools, and the soundness of the whole scheme depends on the correct cooperation between all tools. Therefore, using SMT solvers in formal verification calls for proof-producing decision procedures, checking algorithm and independent proof checkers.

7.2.1 Proof producing decision procedures

The area of proof-generating decision procedure has been pioneered by Boulton for the HOL system [Bou94] and Necula for Proof Carrying Code [Nec98]. In the context of the latter, the Touchstone theorem prover [NL00] generates LF proof terms.

Several authors have examined UF proofs [dMRS05, NO05]. They extend a pre-existing decision procedure with proof-producing mechanism without degrading its complexity and achieving a certain level of irredundancy. However, their notion of proof is reduced to unsatisfiable cores of literals—to be used to accelerate proof search—rather than proof trees. Chapter 9 presents different proof-format and verifier algorithms that can be defined as extensions of these works.

Modern SMT solvers (*e.g.,* CVC3 [BT07], veriT [BODF09], Z3 [dMB08c] or Yices [DdM06]) are able to automatically discharge formula of industrial size combining various logic fragments such as linear (real or integer) arithmetic, the theory of uninterpreted function symbols or the theory of arrays. The SMT-LIB 2.0 format [STB10] is a standard interface for SMT solvers. It provides a unified syntax for SMT problems and a rich interface for interacting with SMT solvers. The command `check-sat` tests the satisfiability of the problem and is the minimal information that is expected from a SMT solver. More advanced features are unsat cores (`get-unsat-core`) or models (`get-model`).

In case the problem is unsat, the command `get-proof` outputs a *proof* of this fact. The answer to the `get-proof` command is unspecified and is therefore prover-specific. And naturally, the SMT solvers CVC3, veriT and Z3 all use a different syntax and semantics for their proofs. Moreover, the granularity of the proofs greatly differ, *i.e.*, many rules of the SMT solvers proof languages reflect the internal reasoning with various levels of precision. Certain rules detail each computation step, some others account for complex reasoning with no further details. Despite on-going efforts, there is no standard SMT proof format, as SMT-LIB concentrate on providing a standard for the input of SMT solvers.

7.2.2 Proof verification

Independent proof checking. The verification of SAT solvers' results developed as soon as they became efficient enough to tackle industrial size formulae, *e.g.*, formulae coming from the verification of industrial code. At

first, checking algorithms were developed along side a particular solver, *e.g.*, in the works of Goldberg and Nivikov [GN03], who developed BerkMin [GN07], Zhang and Malik [Zha03], for Chaff [MMZ+01], and Biere and Sinz [SB06], for PicoSAT [Bie08]. The problem of independently checking proofs produced by different SAT solvers was attacked by Van Gelder [Van02, Van07] and lead to the verified-unsatisfiable track of the SAT–2005 and SAT–2007 solver competitions. The proof-system is based on the resolution deduction system, such that modern SAT solving algorithm's output can be converted easily to the proof-format. To be able to verify extremely large proofs, Van Gelder proposed a log-space checking algorithm [Van12], which manipulates *explicit* resolution, thus is simple enough to be verified—in theory at least—and trusted.

Proof reconstruction. During the past few years, interactive proof assistants have been very successful in the domain of software verification and formal mathematics. In these areas the amount of formal proofs is impressive. For Coq [BC04], one of the mainstream proof assistants, it is particularly impressive to see that so many proofs have been done with so little automation. In his POPL'06 paper on verified compilation [Ler06, Section 6, § 7], Leroy gives the following feedback on his use of Coq:

> Our proofs make good use of the limited proof automation facilities provided by Coq, mostly eauto (Prolog-style resolution), omega (Presburger arithmetic) and congruence (equational reasoning). However, these tactics do not combine automatically and significant manual massaging of the goals is necessary before they apply.

Proof reconstruction is an approach to alleviate this lack of automation using proof producing ATPs: if its proofs can be reconstructed in a proof assistants and checked by, the ATP benefits from the soundness guaranties provided by the proof assistant, and the proof assistant benefits from the automation provided by the ATP.

Several approaches have been proposed to integrate new decision procedures in proof assistants for various theories. First-order provers have been integrated in Isabelle [PS07], HOL [Hur99] or Coq [CC05]. These works rely generally on resolution proof trees. Similar proof formats have been considered to integrate Boolean satisfiability checking in a proof assistant. Armand *et al.* [AGST10] have extended the Coq programming language with machine integers and persistent array and have used these new features to directly program and prove sound, in Coq a reflexive SAT checker. On a similar topic, Weber and Amjad [WA09] have integrated a state-of-the-art SAT solver in Isabelle/HOL, HOL4 and HOL Light using translation from SAT resolution proofs to LCF-style proof objects.

The proof reconstruction approach has also been applied to SMT solvers. McLaughlin *et al.* [MBG06] have combined CVC Lite and HOL light for quantifier-free first-order logic with equality, arrays and linear real arithmetic. Ge and Barrett have continued that work with CVC3 and have extended it to quantified formulae and linear integer arithmetic. This approach highlighted the difficulty of proof reconstruction. Independently, Fontaine *et al.* [FMM+06] have combined HARVEY with ISABELLE/HOL for quantifier free first-order formulae with equality and uninterpreted functions. In their scheme, ISABELLE solves UF sub-proofs with hints provided by HARVEY.

Böhme and Weber [BW10] developed a proof reconstruction approach for Z3 proofs in the theorem provers ISABELLE/HOL and HOL4. Their implementation is particularly efficient but their fine profiling shows that a lot of time is spent re-proving sub-goals for which the Z3 proof does not give sufficient details. The integration of Z3 in ISABELLE/HOL is now part of SLEDGEHAMMER [PS07], a meta prover, or interface, that brings the automation of different ATPs, including SMT solver [BBP11], to ISABELLE/HOL. SLEDGEHAMMER has been a powerful tool to discharge ISABELLE/HOL proof goals [BN10], and associated with counterexample generators, makes proving in ISABELLE "more enjoyable and productive"[BBN11].

The COQ proof assistant currently lacks a tool as powerful as SLEDGEHAMMER, but recent works have improved the situation. Most notably, Armand *et al.* [AFG+11] have extended their previous work [AGST10] to check proofs generated by the SMT solver VERIT [BODF09]. Independently, we proposed a proof-format for SMT solvers and a verifier implemented in COQ [BCP11], described in Chapter 8.

Chapter 8

Result certification of Satisfiability Modulo Theory solvers inside Coq using a reflexive verifier

The present chapter details our result certification approach for Satisfiability Modulo Theory (SMT) solvers, to i) integrate SMT solvers inside Coq and ii) alleviate the problem of including Automated Theorem Provers (ATPs) into the Trusted Computing Base (TCB) of our static analyses result certification approach. First, Section 8.1 presents some specificity of Coq regarding result certification and proof checking, and gives an overview of our approach. Then, Section 8.2 details a proof system adapted to SMT solvers and Section 8.3 presents a reflexive proof verifier. Finally, Section 8.4 describes the proof-generation scheme and preliminary experimental results, and Section 8.5 concludes with a discussion on further work, and a comparison between our approach and another SMT proofs checker for Coq.

8.1 Satisfiability Modulo Theory result certification in Coq

8.1.1 Result certification in Coq

There are two main methods for integrating computations in formal proofs, *e.g.*, a new decision procedure in a system like Coq: the *autarkic* and the *sceptical* approaches.[1] The sceptical approach relies on an external tool, written in an other programming language than Coq, that builds a Coq

[1] We rephrase here the definitions introduced by Barendregt and Barendsen [BB02] in the context of formal proofs in Coq.

proof term for each formula it can prove. The main limit of this approach is the size of the exchanged proof term, especially when many rewriting steps are required [GM05]. The autarkic approach requires to *verify the prover* by directly programming it in Coq and mechanically proving its soundness. Each formula is then proved by running the prover inside Coq. Such a *reflexive* approach [GM05] leads to short proof terms but the prover has to be written in the constrained environment of Coq. Programming a state-of-the-art SMT solver in a purely functional language is by itself a challenging task; proving it correct is likely to be impractical—with a reasonable amount of time.

The result certification methodology suggests a trade-off between the two previous extreme approaches: programming a reflexive verifier that uses certificates—or hints—given by an untrusted prover programed in any efficient programming language. Such an approach has been successfully applied to numerous decision procedures [GM05, Bes07, AGST10, Ler06], and benefit from the efficiency of the Coq reduction engine [GL02], that allows the evaluation of Coq programs with the same efficiency as OCaml programs. Using a verifier programed in Coq has the following advantages: 1) The verifier is simpler to program and prove correct in Coq than the prover itself; 2) Termination is obtained for free as the number of validation steps is known beforehand; 3) The certificates convey the minimum amount of information needed to validate the proofs and are therefore smaller than genuine proof terms. This last point is especially useful when a reasoning takes more time to explain than the time to directly perform it in the Coq engine, *i.e.*, when the proof terms are huge but the decision procedure is efficient in Coq. This design provides a good trade-off between proof time checking and proof size.

8.1.2 Overview of the approach

Programming a reflexive verifier inside Coq paves the way for the integration of different tools, providing they all can generate the appropriate certificates. This makes the approach particularly relevant to the problem of excluding ATPs from the Trusted Computing Base (TCB) of our static analysis result certification approach, as experiments have shown that committing to a particular solver would not allow to discharge all the necessary verification conditions.

A generic proof system. To ensure the format of the certificates will be relevant to different tools, we propose a proof format derived from the *lazy* SMT approach described in Chapter 7, which we see as a common denominator of efficient SMT solving algorithms. This presentation of SMT solving establish a clear separation between propositional reasoning on one hand and theory reasoning on the other. Following this separation, the proof-system,

presented in Section 8.2, isolates a propositional proof of unsatisfiability and a set of theory lemmas that correspond to the conflict clauses. This allows us to benefit from existing approaches to the verification of SAT proofs, and to reuse the implementations they lead to, such as the SAT verifier programed in Coq by Armand *et al.* [AGST10].

An upgradable proof checker. Checking the result of a new analysis may require a richer theory of semantic states, and introduce new abstract domains. Therefore, the proof checker must be extensible to new theories, and account for non-convex ones. To allow for such extensions, the proof-system models the Nelson-Oppen equality exchange with case-split, as presented in Section 8.2.2.[2] This allows us to design a modular proof checker that combines *theory specific* checkers. As long as a checker respects the interface presented in Section 8.3.1, adding a new theory does not require any modification of the rest of the implementation nor of the proofs attached to it.

Beside leaving room for possible extensions, the modular design of the proof checker allows the replacement of a theory specific checker by a new implementations. This open the possibility for some theories to benefit from specific optimised checking algorithms. For example, Chapter 9 presents different checking algorithms for the theory of equality and Uninterpreted Functions (UF), that exhibit different behaviours. Moreover, SMT solvers may pay a high price for giving extensive details of their reasoning, and may omit large theory reasoning from their trace. In such cases, it may be more practical to reuse a decision procedure already implemented in Coq, if it exists.

Assessing the viability of the approach. To assess the viability of the proof system and the efficiency of the proof checker, without having to rely on the output of the current implementation of SMT solvers, we have developed our own certificate generator for conflict clauses. However, the SMT-LIB standard does not provide a "`get-conflict` " command, and SMTs in general do not currently provide the conflicts clauses, so we have implemented a lazy SMT proof search, presented in Section 8.4.1, that outputs the necessary conflicts using black box SMTs as a theory engine and a SAT engine. Our experiments, described in Section 8.4, indicate that unsat cores are relatively small, and their proofs are obtained with a modest overhead by our hand-crafted proof-producing prover. This methodology allows us to implement a prover co-designed with the Coq verifier, which

[2]Optimisation of the SMT solving algorithm that do not rely on the NO algorithm for the combination of theories, *e.g.*, Delayed Theory Combination [BCF+06] and Model Based Theory Combination [dMB08a], can be understood as relying on the SAT engine for the combination, and modelling their behaviour with conflict clauses should be possible, but is left as further work.

therefore has the advantage of generating the exact level of details needed to validate the proof.

Contributions. The contributions of the work described in the present chapter can be summarised as follows:

- A new methodology for exchanging unsatisfiability proofs between an untrusted SMT solver and a proof assistant with computation capabilities like Coq. Our proof format is modular: it separates propositional reasoning from theory reasoning, and structures the communication between theories using the Nelson-Oppen combination scheme.
- A modular reflexive Coq verifier that allows for fine-tuned theory specific verifiers exploiting as much as possible the efficient Coq reduction engine. The current verifier is able to verify proofs for quantifier-free formulae mixing linear arithmetic and uninterpreted functions.
- A proof-generation scheme that uses state-of-the-art SMT solvers in a black-box manner, and only requires the SMT solvers to extract unsat-cores and propositional models—features that have been standardised by the SMT-LIB 2 format.

8.2 Proof system

To describe the proof-checker without getting into the details of its programming, which relies heavily on dependent types, and to present the expected proof format, we first formalise the proof system underlying our approach.

8.2.1 S.M.T. proof format

A witness of unsatisfiability of the input formula is given by a proof of unsatisfiability of a propositional formula composed of the propositional abstraction of the input formula, plus conflict clauses that correspond to negations of unsatisfiable multi-theory conjunctions. The two parts of the reasoning are merged using the proof rule presented in Figure 8.1. A judgement of the form $\Gamma \vdash \mathit{cert} : F$ means that formula F can be deduced from hypotheses in Γ, using certificate *cert*.

$$\frac{\begin{array}{c} f^{\mathbb{B}}, \neg C_1^{\mathbb{B}}, \ldots, \neg C_n^{\mathbb{B}} \vdash_{Boolean} \mathit{cert}^{\mathbb{B}} : \text{False} \\ \forall i = 1, \ldots, n,\ \sigma(C_i^{\mathbb{B}}) \vdash_{NO} \mathit{cert}_i : \text{False} \end{array}}{\sigma(f^{\mathbb{B}}) \vdash_{SMT} \Big(\sigma, (\mathit{cert}^{\mathbb{B}} : f^{\mathbb{B}}), \big[(\mathit{cert}_1 : C_1^{\mathbb{B}}), \ldots, (\mathit{cert}_n : C_n^{\mathbb{B}})\big]\Big) : \text{False}}$$

Figure 8.1: Proof rule merging propositional and theory reasoning

In the judgement $\sigma(f^{\mathbb{B}}) \vdash_{SMT} cert$: False, the certificate *cert* is composed of three elements: a mapping σ between propositional variables and theory literals, a propositional abstraction $f^{\mathbb{B}}$ of F and a list $C_1^{\mathbb{B}}, \ldots, C_n^{\mathbb{B}}$ of conjunctions of propositional variables. For this judgement to establish that the ground formula F is unsatisfiable, several premises have to be verified by the reflexive checker. First, $\sigma(f^{\mathbb{B}})$ must be reducible to F. It means that the propositional abstraction is just proposed by the untrusted prover and checked correct by the reflexive verifier. Then, the conjunction of $f^{\mathbb{B}}$ and all the negation $\neg C_1^{\mathbb{B}}, \ldots, \neg C_n^{\mathbb{B}}$ must be checked unsatisfiable with a propositional verifier. This verifier can be helped with a dedicated certificate $cert^{\mathbb{B}}$—*e.g.*, taking the form of a refutation tree.[3] At last, every multi-theory conjunction $\sigma(C_i^{\mathbb{B}})$ must be proved unsatisfiable with a dedicated certificate $cert_i$. This is done with the judgement $\vdash_{NO}$ which is explained in the next subsection. Figure 8.2 presents the certificate for the running example. It is composed of the mapping (7.2), the propositional abstraction (7.1) together with an propositional unsatisfiability certificate, and the two conjunctions corresponding to the only two conflict clauses needed to prove unsatisfiability, together with their respective multi-theory certificates.

---

a) Mapping σ (7.2) from propositional variables to multi-theory atoms:

A	B	C	D	E
$f(f(x) - f(y)) \neq f(z)$	$x \leq y$	$y + z \leq x$	$z \geq 0$	$y - z \leq x$

b) Propositional abstraction (7.1) of the initial formula and SAT certificate:

- $f^{\mathbb{B}} = A \wedge B \wedge ((C \wedge D) \vee (E \wedge \neg D))$
- $cert^{\mathbb{B}}$ (unsatisfiability of $f^{\mathbb{B}} \wedge \neg C_1^{\mathbb{B}} \wedge \neg C_2^{\mathbb{B}}$)

c) Theory lemmas and certificates:

- $(cert_1^{NO}\ ,\ C_1^{\mathbb{B}} = A \wedge B \wedge C \wedge D)$ corresponding to the conflict clause (7.5)
- $(cert_2^{NO}\ ,\ C_2^{\mathbb{B}} = B \wedge \neg D \wedge E)$

Figure 8.2: SMT certificate for the running example formula presented in Figure 7.1, the sub-certificates $cert^{\mathbb{B}}$, $cert_1^{NO}$, and $cert_2^{NO}$, are not instantiated to keep the chapter readable

[3] As explained in introduction, we do not focus on this specific part. We instead rely on the reflexive tactic proposed by Armand *et al.,* [AGST10, AFG+11].

8.2.2 A proof system for Nelson-Oppen with case-split

Theory exchange is modelled by the Nelson-Oppen proof rule presented in Figure 8.3. We assume here a collection of n theories $T_1, \ldots, T_n$. In this judgement, Γ_i represents an environment of pure literals of theory T_i. Each theory is equipped with its own deduction judgement $\Gamma_i \vdash_{T_i} cert_i : (\Gamma'_i, eqs)$ where Γ_i and Γ'_i are environments of theory T_i, $cert_i$ is a certificate specific to theory T_i and *eqs* is a disjunction of equalities between variables. Such a judgement reads as follows: assuming that all the literals in Γ_i hold, we can prove (using certificate $cert_i$) that all the literals in Γ'_i hold and deduce a disjunction of equalities *eqs*. The disjunction *eqs* can then be used by following deductions.

$$\frac{\begin{array}{c}\Gamma_i \vdash_{T_i} cert_i : (\Gamma'_i, \bigvee_{k=1}^{m} x_k = y_k) \\ \bigwedge_{k=1}^{m} \Big(\Gamma_1 :: [x_k = y_k], \ldots, \Gamma'_i, \ldots, \Gamma_n :: [x_k = y_k] \vdash_{NO} sons[k] : \text{False}\Big)\end{array}}{\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (cert_i, sons) : \text{False}}$$

Figure 8.3: Proof rule modelling a Nelson-Oppen equality exchange

Figure 8.4 presents a certificate for the Nelson-Oppen proof rule represented as a *tree*. The array *sons* contains certificates for three equalities: $(cert_{i_1}, sons_1)$, $(cert_{i_2}, sons_2)$, $(cert_{i_3}, sons_3)$. The content of the arrays $sons_1$ and $sons_3$ are not represented, but the array $sons_2$ contains two certificates $(cert_{i_{21}}, sons_{21})$ and $(cert_{i_{22}}, sons_{22})$, and so on and so forth until certificates with empty arrays, which constitute the leaves of the tree.

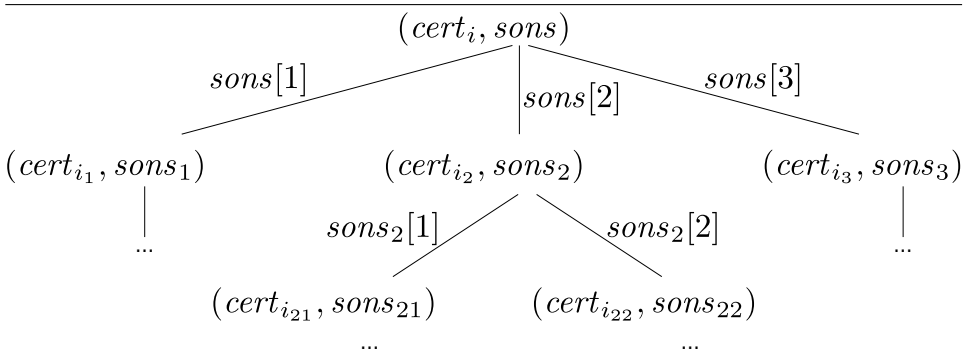


Figure 8.4: Example of Nelson-Oppen certificate seen as a tree

Such tree-like certificates are checked using the Nelson-Oppen proof rule, presented in Figure 8.3. The judgement $\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (cert_i, sons) : \text{False}$ holds only if, given an environment $\Gamma_1, \ldots, \Gamma_n$ of the joint theory $T_1 + \ldots + T_n$, the certificate $(cert_i, sons)$ allows to exhibit a contradiction, *i.e.*, False.

Suppose that the disjunction *eqs* is empty, *i.e.*, trivially *false*. The certificate $cert_i$ establishes a judgement of the form $\Gamma_i \vdash_{T_i} cert_i : (\Gamma'_i, \text{False})$,

and the proof rule can be simplified as follows.

$$\frac{\Gamma_i \vdash_{T_i} cert_i : (\Gamma'_i, \text{False})}{\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (cert_i, \emptyset) : \text{False}}$$

The conjunction over the k certificates in *sons* is empty, *i.e.*, trivially *true*, and therefore can be omitted. This simpler rule formalises a proof that Γ_i is contradictory, therefore the joint environment $\Gamma_1, \ldots, \Gamma_n$ is contradictory, and the judgement holds.

An important situation is when the disjunction *eqs* is limited to a single equality during a proof. This corresponds to the case of convex theories—*e.g.*, LRA and UF—for which the Nelson-Oppen algorithm never needs to perform case-splits [NO79]. In this case, the array of certificate *sons* contain only one certificate *son*—singular—if any, therefore the tree-form of the certificate degenerates into a list, and the proof rule can be simplified as follows.

$$\frac{\begin{array}{c}\Gamma_i \vdash_{T_i} cert_i : (\Gamma'_i, x = y) \\ \Gamma_1 :: [x = y], \ldots, \Gamma'_i, \ldots, \Gamma_n :: [x = y] \vdash_{NO} son : \text{False}\end{array}}{\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (cert_i, son) : \text{False}}$$

In the general case, the rule presented in Figure 8.3 applies: we recursively exhibit a contradiction for each equality $(x_k = y_k)$ using the kth certificate of *sons*, *i.e.*, $sons[k]$ for a joint environment

$$\Gamma_1 :: [x_k = y_k], \ldots, \Gamma'_i, \ldots, \Gamma_n :: [x_k = y_k]$$

enriched with the equality $(x_k = y_k)$.[4] The judgement holds if all the branches of the case-split over the equalities in *eqs* lead to a contradiction.

For the example of Nelson-Oppen equality exchange presented in Figure 7.5, we start with the sets Γ_{LRA} and Γ_{UF} of LRA hypotheses (resp. UF hypotheses). LRA and UF are convex theories, therefore the certificate $cert_1^{NO}$ presented below is a list of LRA or UF certificates, not a tree like the certificate described in Figure 8.4.

$$cert_1^{NO} \doteq (cert_1^{\text{LRA}}, \{(cert_1^{\text{UF}}, \{(cert_2^{\text{LRA}}, \{(cert_2^{\text{UF}}, \{\})\})\})\})$$

A first certificate $cert_1^{\text{LRA}}$ is required to prove the equality $x = y$, then a certificate $cert_1^{\text{UF}}$ to prove $t_3 = t_5$, then a certificate $cert_2^{\text{LRA}}$ to prove the equality $t_6 = z$, and at last a certificate $cert_2^{\text{UF}}$ to find a contradiction.

Discharging unsat mono-theory conjunctions. Each part of the NO proof is theory-specific: each theory must justify either the equalities exchanged or the contradiction found. A LRA proof of $a = b$ is made of two

[4] The rule use the notation of lists for an enriched certificate $\Gamma :: [x_k = y_k]$, but adding an equation to a *pure* environment can be implemented differently, *e.g.*, using memory space occupied by equations that are non longer necessary, or a different data-structure.

Farkas proofs [Sch98] of $b-a \geq 0$ and $a-b \geq 0$. Each inequality is obtained by a linear combination of hypotheses that preserves signs. For example, the previous certificate $cert_1^{\mathrm{LRA}}$ explains that hypothesis (7) gives $y-x \geq 0$ and $(8)+(9)$ gives $x-y \geq 0$. For more details on the proof-system and on certificates for LRA, see Appendix A. A UF proof of $a=b$ is made of a sequence of rewrites that allows to reach b from a. For example, the certificate $cert_1^{\mathrm{UF}}$ explains the equality $t_3 = t_5$ with the following rewritings:

$$t_3 \xrightarrow{\text{trans. with (1)}} f(y) \xrightarrow{\text{congr. with (11)}} f(x) \xrightarrow{\text{trans. with (2)}} t_5$$

For more on this proof format, alternatives and comparison, see Chapter 9.

8.3 Reflexive S.M.T. proof checker in Coq

This section presents the design of a reflexive Coq verifier for a Nelson-Oppen style combination of theories. Section 8.3.1 presents the main features of the theory interface, and Section 8.3.2 explains the data-structures manipulated by the Nelson-Oppen proof-checker, *i.e.*, its dependently typed environment and its certificates.

8.3.1 Theory interface

A theory `T` defines a type `sort` for sorts , `term` for terms, and `form` for formulae. Sorts, terms and formulae are equipped with interpretation functions `isort`, `iterm` and `iform`. The function `isort:sort→`**`Type`** maps a sort to a Coq type. This allows us to design a Nelson-Oppen verifier that never needs to be modified, and whose proofs will not need to be updated when adding a new theory: new sorts are *explained* in the interface of the new theory checker.

Terms and formulae are interpreted with respect to a typed environment `env ∈ Env` defined by

```
Env := var → ∀ (s:sort), isort s
```

Each theory uses an environment $\Gamma \in$`Gamma` to store formulae, equipped with an interpretation function `ienv`. Listing 8.1 on the following page presents the complete API expected of environments. The `empty` environment represents an empty conjunction of formulae, *i.e.*, the assertion *true* and is such that `ienv env empty` holds for any environment. The operation `add` models the addition of a formula and is compatible with the interpretation `iform` of formulae. Our instantiations exploit the fact that environments are kept abstract: for UF, environments are radix trees allowing a fast look-up of formulae; for LRA, they are simple lists but arithmetic expressions are normalised (put in Horner normal form) by the `add` operation.

```
Record GammaAPI : Type :=
{|
  empty : Gamma ; add : form → Gamma → Gamma;
  ienv : Env → Gamma → Prop;
  ienv_empty : ∀ env, ienv env empty;
  ienv_add : ∀ (f : form) (s : Gamma) (env : Env),
     ienv env s → iform env f → ienv env (add f s)
|}.
```

Listing 8.1: Environment API definition

A representative theory record in presented in Listing 8.2. The key feature provided by a theory T is a proof-checker `Checker`. It takes as argument an environment Γ and a certificate *cert*. Upon success, the checker returns an updated environment Γ' and a list

$$eqs = (x_1 =_{s_1} x'_1, \ldots, x_n =_{s_n} x'_n)$$

of equalities between sorted variables. In such cases, `Checker_sound` establishes that $\Gamma \vdash_T cert : (\Gamma', eqs)$ is a judgement of the Nelson-Oppen proof system presented in Section 8.2.2. The environment Γ' denotes the memory used for further computations, and can be used to store intermediary results, potentially useful for the following deductions. And as the type `Env` is abstract in the interface, each theory checker can defined environments optimised for the theory.

```
Record Thy :=
{|
   sort : Type; term : Type;  form : Type;
   sort_of_term : term → sort;  isort : sort → Type;
   Env := var → ∀ (s:sort), isort s;
   iterm : Env → ∀ (t : term), isort (sort_of_term t);
   iform : Env → form → Prop
   ...
   Checker : Gamma → Cert → option(Gamma * list (Eq.t sort))
   Checker_sound :
     ∀ cert Γ Γ′ eqs, Checker Γ cert = Some(Γ′, eqs)
         → ∀ (env : Env),  ienv env Γ
         →  ienv env Γ′
            ∧  ∃ s, ∃ x, ∃ y, (x =s y) ∈ eqs ∧ env x s = env y s
|}.
```

Listing 8.2: Theory record definition

8.3.2 Nelson-Oppen proof checker

Given a list of theories $\mathtt{T}_1, \ldots, \mathtt{T}_n$ the environment of the Nelson-Oppen proof-checker is a dependently typed list such that the ith element of the list is

an environment of type $\texttt{T}_i$`.(Gamma)`. Dependently typed lists are defined as follows:

```
Inductive dlist (A : Type) (typ : A → Type) : list A  →  Type :=
| dnil : dlist A typ nil
| dcons : ∀ (x : A) (e : typ x) (lx : list A) (le : dlist lx),
                      dlist A typ (x::lx).
```

A term `dcons x e lx le` constructs a list with head `e` and tail `le`. The type of `e` is `typ x` and the type of the elements of `le` is given by `(List.map typ lx)`. A term `dnil A typ nil` constructs the empty list. It follows that the environment of the Nelson-Oppen proof-checker has type:

$$\texttt{dlist Thy Gamma (T}_1\texttt{::...::T}_n\texttt{)}$$

A single proof-step consists in checking a certificate `JCert` of the joint theory defined by

$$\texttt{JCert := T}_1\texttt{.(Cert) + ... + T}_n\texttt{.(Cert)}$$

Each certificate triggers the relevant theory proof-checker and derives an eventually empty list of equalities, *i.e.*, a proof of unsatisfiability. Each equality $\texttt{x} =_s \texttt{y}$ is cloned for each sort `s'` such that `isort s = isort s'` and propagated to the relevant theory. Each equality of the list is responsible for a case-split that may be recursively closed by a certificate, as presented in Section 8.2.2. A certificate for the Nelson-Oppen proof-checker is therefore a tree of certificates defined by:

```
Inductive Cert := Mk (cert : JCert) (lcert : list Cert).
```

The Nelson-Oppen verifier consumes the certificate and returns `true` if the last deduced list of equalities is empty. In all other cases, the verification aborts and the verifier returns `false`.

8.4 Experiments

The purpose of our experiments is twofold. They show that our SMT format is viable and can be generated for a substantial number of benchmarks, and they assess the efficiency of our Coq reflexive verifier.

8.4.1 Certificate generation using collaborating black-box solvers

To generate our SMT proof format, we implement the simple lazy SMT loop discussed in Section 7.1, using SMT-LIB 2 scripts to interface with off-the-shelf SMT solvers. The SMT-LIB 2 [STB10] standard exposes a rich API for SMT solvers that makes this approach feasible. More precisely, SMT-LIB 2 defines scripts that are sequence of commands to be run by SMT solvers.

The `asserts f` command adds the formula `f` to the current context and the `check-sat` command checks the satisfiability of the current context. If the context is satisfiable—`check-sat` returns `sat`—the `get-model` command returns a model. Otherwise, the `get-unsat-core` command returns an unsat core, *i.e.*, a minimised unsatisfiable subset of the current context.

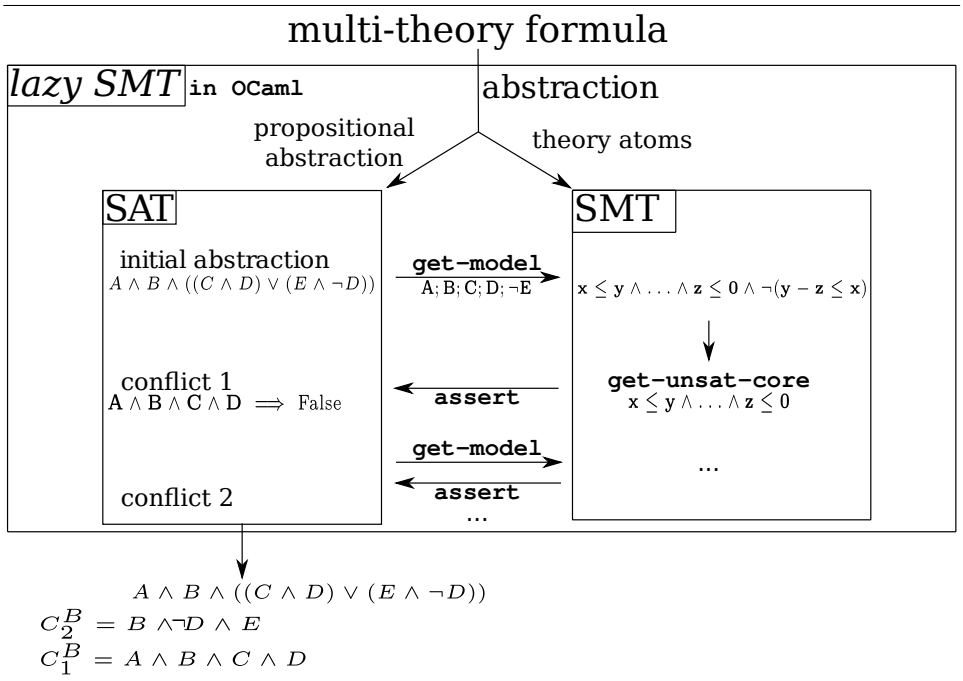


Figure 8.5: Generation of conflict clauses, using the lazy SMT scheme presented in Figure 7.4, implemented using SMT solvers as black-boxes, with the SMT-LIB API—commands of the API are typed using a `typewriter` font

The SMT loop, as presented in Figure 8.5, is implemented using SMT-LIB 2 compatible off-the-shelf SAT and SMT solvers (we chose Z3 for both). Given an initial unsatisfiable formula, the protocol is the following. To begin with, the propositional abstraction of the input formula is computed and sent to the SAT solver by a simple front-end programed in OCaml. For each propositional model returned by the SAT solver, the SMT solver is asked for an unsat core, whose negation is sent to the SAT solver as a conflict clause. The loop terminates when the SAT solver returns an `unsat` status. Once all the unsat cores have been discovered, our OCaml prover generate certificates for them, following the proof system described in Section 8.2.2. This untrusted certifying prover implements the Nelson-Oppen algorithm [NO79] described in Chapter 7. Overall, unsat cores tend to be very small (10 literals on average) and therefore our certifying prover

is not the bottleneck. The propositional proof is obtained by running an independent certifying SAT solver. Unlike SMT solvers, DPLL-based SAT solvers have standardised proofs: *resolution proofs*.

Our prototype could be optimised in many ways. For instance, a propositional proof could be obtained directly without re-running a SAT solver. Our scheme would also benefit from a SMT-LIB 2 command returning all the theory lemmas (unsat cores are only a special kind of those) needed to reach a proof of unsatisfiability.

8.4.2 Sampling

Evaluation was performed on quantifier-free first-order unsatisfiable formulae over the combinations of the theory of equality and uninterpreted functions (UF), linear real arithmetic (LRA), linear integer arithmetic (LIA) and real difference logic (RDL). All problems were taken form the SMT-LIB benchmarks and we have limited our evaluation to problems independently solved under 30 seconds by the SMT solver used in our search. Some category being larger than others (there are more than 4000 unsatisfiable benchmarks in UF for example), we have focused our evaluation on diversity and are working through the largest categories. These benchmark are not meant be representative of goals arising in Coq proofs, but are a combination of very large and fairly hard formulae designing to stress test SMT solvers, and should allow us to test the limits of our proof checker, and to assess its efficiency. Further testing involving typical Coq goals would be welcomed, but such benchmarks does not exist for the moment, and crafting a reasonable amount of sufficiently diverse Coq goals is left to further work.

8.4.3 Results

Efficiency. Table 8.1 on the next page shows our results sorted by logic. For each category, we measure the average running time of Z3[5] (Solved), the average running time of our certificate generation (Generation), and the average time spent checking the certificates (Checking). The *Solved* time can be seen as a best-case scenario: the certifying prover uses Z3 and provide proofs that can be checked in Coq, so we do not expect faster results than the standalone state-of-the-art solver. We also measure the time it takes Coq to type-check our proof term (Qed) and have isolated the time spent by our Coq reflexive verifier validating theory lemmas (Thy). The generation phase (Generation) and the checking phases (Checking) have an individual timeout of 150 seconds. These timeouts account for most of the failures, the remaining errors come from shortcomings of the prototype.

As the success rates indicates, our simple SMT loop may fail to produce certificates before timeout. For UF and RDL we only generate certificates

[5]On its own, natively, on the initial formula.

	Solved (Z3)		Generation		Checking		
Logic	#	Time (s)	Success	Time (s)	Success	Thy (s)	Qed (s)
UF	613	0.96	31.3%	42.55	100%	0.29	16.81
LRA	248	0.65	79.4%	6.79	69.5%	0.28	4.02
UFLRA	407	0.11	100%	0.72	98.8%	0.02	3.56
LIA	401	1.86	74.3%	9.05	46.0%	2.26	7.02
UFLIA	159	0.05	97.5%	8.15	96.1%	0.33	2.91
RDL	79	4.01	38.0%	11.24	53.3%	0.14	3.64
Total	1907	0.87	67.1%	11.02	80.8%	0.45	6.45

Table 8.1: Experimental results for selected SMT-LIB logics

for a third of the formulae. The generation of certificates could be optimised further. A more clever proof search strategy could improve both certificate generation and checking times: smaller certificates could be generated faster and checked more easily. Yet, the bottleneck is the reflexive verifier, which achieves 100% success ratio for UF only. Currently, we observe that our main limiting factor is not time but the memory consumption of the Coq process. A substantial amount of our timeouts are actually due to memory exhaustion. We are investigating the issue, but the objects we manipulate (formulae, certificates) are orders of magnitude larger than those manipulated on a day-to-day basis by a proof-assistant, and we are reaching the limits of the system. As a matter of fact, to perform our experiments we already overcome certain inefficiencies of Coq. For instance, to construct formulae and certificates we by-pass the Coq front-end, which is not efficient enough for this application, and use homemade optimised versions of a few Coq tactics.

Overall, the theory specific checkers account for less then 7% of checking time (0.45s in 6.45s). However, this average masks big differences. For UFLRA, the checker spends less than 1% of its time in the theories, but for the integer arithmetic fragments it represents 11% of checking time for UFLIA and 32% for LIA. For integer arithmetic the success ratio is rather low. It is hard to know whether this is due to the inherent difficulty of the problems or whether it pinpoints an inefficiency of the checker. The fault might also lie on the certifying prover side. In certain circumstances, it performs case-splits that are responsible for long proofs.

Shape of theory reasoning. To understand the differences between benchmarks in the amount of time spent in the theories, and to evaluate the difficulty of certifying the conflict-clauses, we have to look at the number of conflict clauses per formula, and to the size of these clauses.

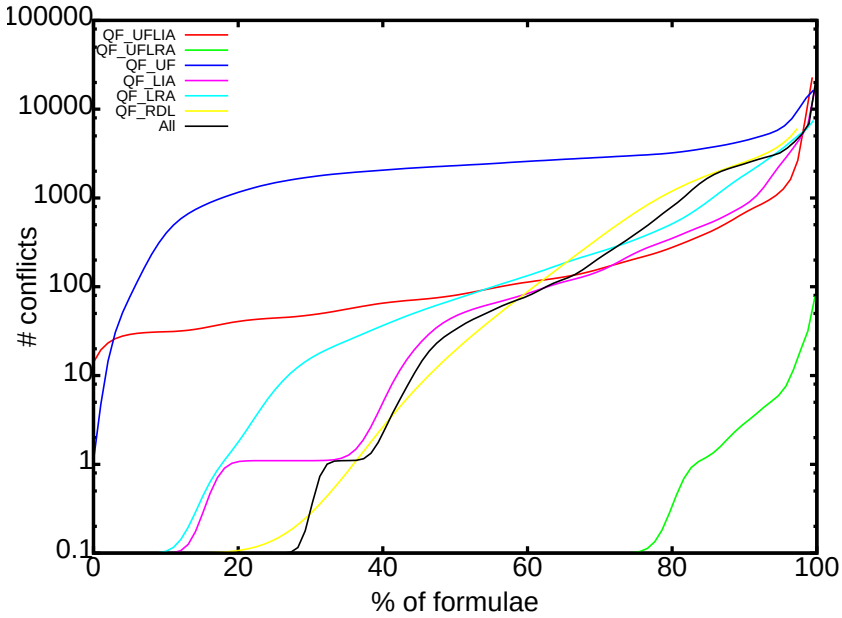


Figure 8.6: Number of conflic-clauses per formula. The x-axis gives the percentage of formulae for which the number of conflict-clauses is under a given number on the y-axis, one curve per theory, and one curve—the black one—for the set of all formulae in all theories.

Figure 8.6 presents a comparison of the number of conflicts between the different theory combinations. Overall, our experiments suggest that in general, formulae do not require a lot of conflict-clauses. On the figure, 62% of all the formulae generates less than 100 conflicts. As can be seen on the figure, 80% of the formulae in the UFLRA fragment—green curve—generate less than 1 conflict, hence their propositional abstraction is unsatisfiable from the start. This explains why the checker spends less than 1% of the time in the theory. For some of these benchmarks, our proof-producing solver actually outrun state-of-the-art solvers, just by checking the satisfiability of the abstraction first. On the other hand, the formulae in the UF fragment—blue curve—generate lots of conflict-clauses, more than 1000 for 80% of them, up to 10000 for a small number of formulae. This can be explained by the fact that a lot of these benchmarks are *crafted* to emphasise inefficiencies in solvers, *e.g.*, not taking into account the symmetry of the formula to simplify the proof search, whereas other benchmarks are generated automatically by verification tools and seam to contain some *noise*.

Figure 8.7 on the next page presents a comparison between the different theory combinations of the average number of literal in the conflict-clause

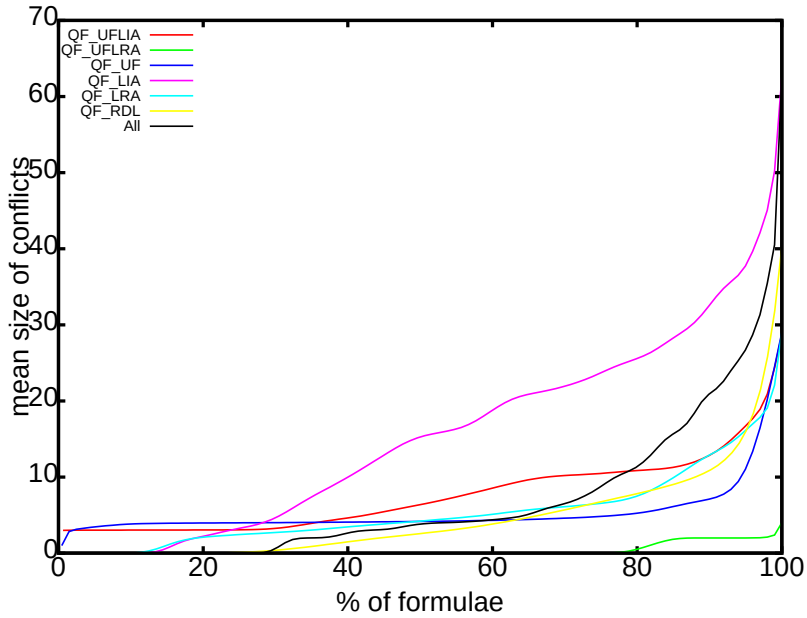


Figure 8.7: Average size of the conflict-clauses. The x-axis gives the percentage of formulae for which the average number of literals in the conflict clauses is under a given number on the y-axis, one curve per theory, and one curve—the black one—for the set of all formulae in all theories.

generated by a formula. Overall, our experiments suggests that conflict-clauses are *small* formulae. On the figure, for 77% of all the formulae the average number literals per conflict-clauses is 10 or less. This corresponds to theory lemmas of *conjunctions* of 10 literals, even for initial formulae with hundreds of literals. This behaviour is consistent for the benchmarks of all the fragments we tested, and suggests that the full power of state-of-the-art SMT solvers is not necessary to prove that a conflict-clause is unsatisfiable. In other words, there is not need for SMT solvers to certify the conflict-clauses, it can be done afterwards by a much simpler proof-producing decision procedure. The only thing that is required from a SMT solver to obtain certificates in our format is to output the set of conflict clauses it used.

8.5 Conclusion

We have presented a reflexive approach for integrating a SMT solver in a proof assistant like Coq. It is based on a SMT proof format that is independent from a specific SMT solver. We believe our approach is robust to changes in the SMT solvers but allows nonetheless to benefit from their

improvements. For most usages, the overhead incurred by our SMT loop is acceptable. It could even be reduced if SMT solvers gave access to the theory lemmas they use during their proof search, and we are confident that such information could be generated by any SMT solver with little overhead.

Implementing our approach requires proof-producing decision procedures for conjunctions of the theory involved, that use a proof format accepted by the theory checkers targeted. However, the hard job is left to the SMT solver that extracts unsat cores. A fine-grained control over the produced proof has the advantage of allowing to optimise a reflexive verifier and of ensuring the completeness of the verifier with respect to the prover. Our Nelson-Oppen Coq verifier is both reflexive and parametrised by a list of theories. This design is modular and easy to extend with new theories. Our prototype implementation is perfectible but already validates SMT formulae of industrial size. Such extreme experiments test the limits of the proof-assistant and will eventually help at improving its scalability.

8.5.1 Comparison with other SMT result certification in Coq

In parallel with our work, Armand *et al.* extended their integration of SAT solvers in Coq [AGST10] to Satisfiability Modulo Theory solvers [AFG$^+$11]. Their proof-format is built on sequences of clause, obtained through resolution, or using axioms of a particular theory. A proof is then a sequence of combinations of clauses, each step being justified by a *small checker*—*e.g.*, a resolution checker, a UF checker or a LIA checker—until the empty clause is reached. They were able to encode proofs returned by the veriT [BODF09] SMT solver into their format, and to discharge formulae from the SMT-LIB [STB10] library, belonging to the unquantified fragment of the theory of Uninterpreted Functions (UF), Linear Integer Arithmetic (LIA) and Integer Difference Logic (IDL).

There are numerous similarities between their approach and ours. Both works built on the SAT result verification scheme in Coq [AGST10], both approaches rely on proof by reflexion, and both approaches are modular, *i.e.*, can be expanded to new theories by providing new checkers for additional theories. However, our proof-format makes a distinction between propositional reasoning and theory reasoning, and includes a rule for multi-theory reasoning. It allows us to discharge formulae belonging to unquantified *combinations* of UF and Linear Real Arithmetic (LRA), or UF and LIA, and not only LIA *or* UF. Moreover, we do not translate proofs output by an SMT solver such as veriT but rely on our own implementation of a certifying Nelson-Oppen combination, and a lazy SMT loop between a SAT solver and a SMT solver to extract conflict clauses. Therefore, we are able to generate certificate in our own format for any SMT solver complying with the SMT-LIB standard, but can not generate certificates for a substantial number of benchmarks, our generation scheme lacking too many optimisations present

in state-of-the-art SMT solvers and generating too many conflicts clauses.

Their approach was designed to integrate efficiently state-of-the-art SMT solvers in Coq, whereas we preferred to avoid relying on the output of the current implementation of SMT solvers. As a result, their approach integrates efficiently the result of veriT for a small set of theories it can solve, whereas we are independent from the SMT solver, therefore we can not generate certificates for some benchmarks but can easily discharge new theories, as long as some SMT solver can generate unsat cores for it, and we can independently generate and check certificates for that theory. To make the best of both approaches, the two implementations could be merged by integrating our multi-theory checker as a *small checker* for their general SMT proof-checker.

8.5.2 Further work

In the future, we plan to integrate new theories such as the theory of arrays and bit-vectors. Another theory of interest is the theory of constructors that would be useful to reason about inductive types. The verification of SMT proofs involving bit-vectors has already been explored [BFSW11] for the Isabelle/HOL and HOL4 systems, but adapting such works to Coq may raise new issues and provide new opportunities. The theories of arrays and constructors are closely related to the UF theory, and the UF proof checker could probably be expanded to deal with them.

The rising trend of integrating decision procedures in Coq calls for extensive, peer reviewed benchmarks. A good starting point for such a library would be large Coq developments, such as the CompCert project [Ler06], and possibly the benchmarks used for the evaluation of the Sledgehammer [BN10]. For benchmarks more related to program verification, the Why3 [BFMP11] platform for deductive verification provides tools to export Verification Conditions (VCs) to Coq, and large Why development may become valuable sources of Coq benchmarks.

Our implementation can not yet be used to discharge verification conditions generated by the result certification of static analyses for two reasons: the integration of SMTs in Coq is not complete enough, and the implementation does not cover enough automated theorem provers .

The integration of SMTs is not complete enough because we do not provide checkers for all the theories required to describe the theory of semantic states, and we only discharge unquantified formulae, whereas even the abstract VCs presented in Chapter 6 require at least to certify axiom instantiation. Besides arrays and constructor, algebraic data-types are necessary, but they are wildly used in Coq, and existing tactics could be used to build a checker. The main obstacle for the certification axiom instantiation should be to obtain enough information from SMT solvers to recognise, after preprocessing, the axioms that are instantiated.

The experiments described in Chapters 5 and 6 have shown that different automated theorem provers are needed to discharge all VCs. We have restricted ourselves to SMT solvers, but TPTP solver are needed too. And beyond that, even if the proof system can account for different SMT solving algorithms, the current certificate generation is based on a naive lazy SMT proof search and does not benefit from all the optimisations of state-of-the-art SMT solvers. We are confident that modern SMT solvers can be modified to output the necessary conflict clauses, and the fact that a significant amount of SMT-LIB benchmarks can be discharged with our naive conflict generation suggests that generating conflict should not disable too many optimisation, but the actual instrumentation remains to be done. Moreover, certifying optimisations performed by SMT solvers during the preprocessing of the formula remains an open issue.

Chapter 9

Result certification of equality with Uninterpreted Functions logic

We have detailed in Chapter 8 a result verification scheme for SMT solvers. Our verifier uses sub-verifiers in charge of the verification of theory reasoning's. Each sub-verifier is specific to a theory, and ignores all the rest of the reasoning involved when proving valid a many sorted formula. This modular design allows us to change the verifier if a new verification algorithm emerges for a theory.

This feature is essential for an upgradable SMT verifier, as even for the simplest logic, such as the quantifier-free logic of equality with Uninterpreted Function symbols (UF), there exist competing proof formats generated by SMT solvers. Several of those formats have been validated in proof-assistants: Z3 proofs can be reconstructed in Isabelle/HOL and HOL4 [BW10]; veriT proofs can be reconstructed in Coq [AFG+11]. Even though the results are impressive, certain SMT proofs cannot be verified because they are too big.

To alleviate this problem, we propose different UF verifiers that require different amounts of information from the solving decision procedure, thus different proof formats. Our first proof format is tightly related to the axioms of the theory, and translates the result of the decision procedure in a sequence of commands, each command corresponding to an axiom of the theory. Our second proof format is made of *proof forests*, the artifact proposed by Nieuwenhuis and Oliveras to extract efficiently unsatisfiable cores [NO05]. Our third proof format is a trimmed down version of the proof forest reduced only to the edges responsible for triggering a congruence. The proof verifiers for these formats implement more and more of the decision procedure, therefore each format is a different trade off between short proofs and simple verifiers. For the sake of comparison, we have also implemented

a Coq verifier for the UF proof format of Z3 [dMB08b] and compared with the existing Coq verifier for the UF proof format of veriT [AFG+11].

Recent works [BW10, AFG+11] show that SMT proofs are big objects and that the bottleneck is usually the proof-assistant. Ideally, SMT proofs should be i) generated by SMT solvers with little overhead, and ii) verified quickly by proof assistants.

Embedding union-find in UF proofs was previously proposed by Conchon *et al.* [CCKL07]. However, in their approach, union-find computations are interleaved with logic inferences that are responsible for an overhead that is absent from our verifiers. The importance of succinct proofs has been recognised by Stump [Stu09] and its Logical Framework with Side Conditions (LFSC). Our proofs are less flexible than LFSC proofs, but are purely computational, and use the principle of proof by *reflection* [BC04, Chapter 16].

First, Section 9.1 describes the standard decision procedure for UF. It is implemented in most SMT solvers and will therefore be used to generate certificates. Section 9.2 presents the different proof formats we introduced, then Section 9.3 describe their implementations, and the results of our experiments, and finally, Section 9.4 concludes the present chapter.

9.1 Equality and Uninterpreted Functions proof producing decision procedure

Nieuwenhuis and Oliveras have proposed a *proof-producing* congruence closure algorithm for deciding UF [NO05]. Their main contribution is an efficient Explain operation which outputs a (small) set of input equations E needed to deduce an equality, say $a = b$. If $a \neq b$ is also part of the input, $E \cup a \neq b$ is a (small) unsatisfiable core that is used by the SMT solver for backtracking. As a result, SMT solvers using congruence closure run a variant of the Explain algorithm—whether or not they are proof producing.

The Explain algorithm is based on a specific data structure: the proof forest. A proof forest is a collection of trees in which each edge $a \to b$ in the proof forest is labelled by a *reason* justifying why the equality $a = b$ holds. A reason is either an input equation $a = b$ or a pair of input equations $a_1(a_2) = a$ and $b_1(b_2) = b$. For the second case, there must be justifications in the forest for $a_1 = b_1$ and $a_2 = b_2$. Figure 9.1 presents an example of proof forest, with two trees, and edges labelled by the reasons its nodes are equal. Two edges are labelled by a couple of input equations: the edge between x_1 and x_4, which was added after $f_2 = f_3$ was proved, and the edge between x_5 and x_6, that was added after $f_1 = f_3$ and $x_2 = x_3$ were proved. All the other edges come directly from input equations.

A recursive version of Explain is given Listings 9.2 and 9.1. The auxiliary functions NearestAncestor and Parent return (if it exists) respectively the nearest

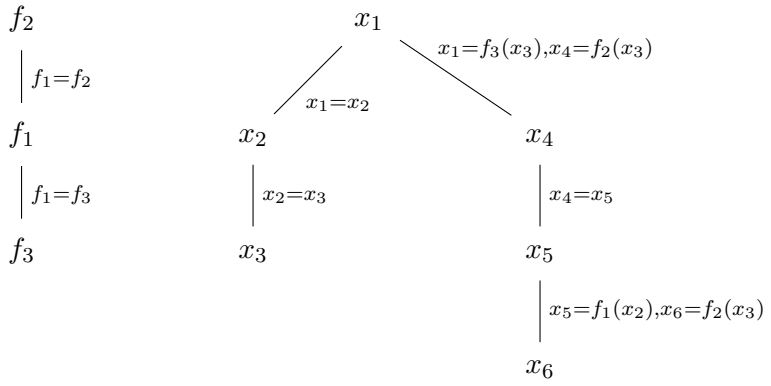


Figure 9.1: Example of proof forest with two trees

common ancestor of two nodes in the proof forest and the parent of a node in the proof forest. A union-find data structure is also used: Union(a,b) merges the equivalence classes of a and b; HighestNode(c) is the highest node (in the proof forest) belonging to the equivalence class of c. When asked for a reason why two variables a and b are equal, the algorithm searches for the nearest common ancestor c of the two nodes, then searches for the highest ancestor c' of c that belongs to its equivalence class, *i.e.*, such that a reason for $c = c'$ has already been given. Then, the algorithm climbs up the path from a to c and from b to c, collecting along the way the input equations responsible for the edges. If it finds an edge between two nodes a' and b', annotated with two input equations $a' = a_1(a_2)$ and $b' = b_1(b_2)$, *i.e.*, an edge between two variables equal because of the congruence axiom, it recursively asks for the reason why $a_1 = b_1$ and $a_2 = b_2$.

9.2 Uninterpreted Functions proof verifiers

The present section details three proof formats that could be placed on a scale from checking of derivation tree in a deductive system to decision procedure that need no certificates. The command verifier would be closer to the checking of a derivation tree, as it apply axioms to verify an equality; the proof forest verifier would be in the middle of the scale, as it requires to implement a union find decision procedure to check that the SMT solver did not make a mistake; the last verifier, that used trimmed forest, would be on the other end of the scale, close to a decision procedure, as it uses the certificates as hints, and re-implements a simpler version of the decision procedure.

```
let ExplainAlongPath(a,c) :=
  a:=HighestNode(a);
  if a = c then return
  else
    b:=Parent(a);
    if edge has form a --a=b--> b
    then  output (a = b)
    else{
      (* edge has form a --b1(b2)=b / a1(a2)=a--> b*)
      output a1(a2)=a and b1(b2)=b;
      Explain(a1,b1);
      Explain(a2,b2)
    };
    Union(a,b);
    ExplainAlongPath(b,c)
```

Listing 9.1: ExplainAlongPath

```
let Explain(a,b) :=
  c:=NearestAncestor(a,b);
  c:=HighestNode(c);
  ExplainAlongPath(a,c);
  ExplainAlongPath(b,c)
```

Listing 9.2: Recursive Explain algorithm

9.2.1 Command verifier

This proof format was used in the experiments presented in Chapter 8. Each command derives new equalities from initial equalities or already derived equalities.

The key commands correspond to the following deduction rules, identical to the axioms defining the theory.

$$\textit{Symmetry}\frac{a=b}{b=a} \qquad \textit{Transitivity}\frac{a=b \quad b=c}{a=c}$$

$$\textit{Congruence}\frac{a=a_1(a_2) \quad b=b_1(b_2) \quad a_1=b_1 \quad a_2=b_2}{a=b}$$

The commands are obtained by running a version of Explain where the union-find structure has been replaced by a hash-table keeping track of already derived equalities. Each call Explain a b appends a *Transitivity* and a *Symmetry* command to the commands obtained by the two successive calls to ExplainAlongPath a c and ExplainAlongPath b c. Each call ExplainAlongPath a c generates a list of commands justifying the equality $a = c$. An edge $a \xrightarrow{a=b} b$ is justified by a command *Hyp* checking that $a = b$ is indeed an input equation.

An edge $a \xrightarrow[a=a_1(a_2)]{b=b_1(b_2)} b$ is justified by a *Congruence* command appended to the result of the recursive calls to Explain a1 b1 and Explain a2 b2 which generate commands justifying the equalities $a_1 = b_1$ and $a_2 = b_2$. The recursive call to ExplainAlongPath b c generates commands justifying the equality $b = c$. The complete list of commands is then obtained by adding a *Transitivity* command to prove $a = c$ from $a = b$ and $b = c$. Each *ExplainAlongPath* call produces a *Transitivity* command on top of either the recursively produced

command in case of an edge labelled by an input equation or a *Congruence*, and the recursively produced commands otherwise.

The verification of such a proof then consists in executing in order each command to derive the wanted equality, checking that each rule is correctly applied.

9.2.2 Proof forest verifier

The Explain algorithm of Listings 9.2 and 9.1, rather than being used to produce certificates, can be turned into a UF proof verifier. The verifier is a version of Explain augmented with additional checks to ensure that the edges obtained from the SMT solver correspond to a well-formed proof forest. The resulting algorithm performs a tree traversal that checks that each edge is correctly annotated.

For instance, the verifier checks that edges are only labelled by input equations. Moreover, for edges of the form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$, the recursive calls to Explain ensure that $a_1 = b_1$ and $a_2 = b_2$ have proofs in the proof forest *i.e.,* a_1 (resp. a_2) is connected with b_1 (resp. b_2) by some valid path in the proof forest. For efficiency and simplicity, the *least common ancestors* are not computed by the verifier but used as untrusted hints. The soundness of the verifier does not depend on the validity of this information as the proposed *least common ancestor* is just used to guide the proof. If the return node is not a common ancestor, the verifier will simply fail.

For this verifier, a UF proof is a pruned proof forest corresponding to the edges walked through during a preliminary run of Explain. As the SMT solver needs to traverse the proof forest to extract unsatisfiable core, we argue that the proof forest is a UF proof that requires no extra-work from the SMT solver.

9.2.3 A verifier using trimmed forests

To avoid traversing the same edge several times, the Explain algorithm and its verifier are using a union-find data structure. Therefore, the Explain verifier implicitly embeds a *decision procedure* for the theory of equality. Our optimised verifier `UFchecker` fully exploits this observation and starts by feeding all the input equalities of the form $a = b$ into its union-find. For the decision procedure, new equalities are obtained by applying the congruence rule and efficiency crucially depends on a clever indexing of the equations. The verifier does not require this costly machinery and takes as argument a trimmed down proof forest reduced to the list of edges of the forest of form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$, as illustrated by Figure 9.2, which presents the trimmed version of the previous example of proof forest. The edge labels indicate the equations that need to be paired to derive $a = b$ by the congruence rule.

Figure 9.2: Example of trimmed forest, obtained with the proof forest presented in Figure 9.1

The algorithm of the verifier is presented in Listing 9.3, where the predicate isEqual(a,b) checks whether a and b have the same representative in the union-find *i.e.*, Find(a) = Find(b). Once again, a preliminary run of Explain is sufficient to gather all the edges arising from the application of the congruence axiom, thus to generate a proof that this verifier can check.

```
let UFchecker input edges (x,y) :=
   for (a = b) ∈ input do Union(a,b) done
   for a --b1(b2)=b / a1(a2)=a--> b in edges do
      if (a1(a2)=a) ∈ input & (b1(b2)=b) ∈ input
                & isEqual(a1,b1) & isEqual(a2,b2)
      then Union(a,b)  else fail
   done
   return isEqual(x,y)
```

Listing 9.3: Verifier algorithm for trimmed forests

9.3 Implementation and experiments

9.3.1 U.F. verifiers in Coq

Our verifiers are implemented using the native version of Coq [BDG11] which features *persistent* arrays [CF08]. Persistent arrays are purely functional data-structures that ensure constant time accesses and updates of the array as soon as it is used in a monadic way. For maximum efficiency, all the verifiers make a pervasive use of those arrays that allows for an efficient union-find implementation: union and find have their logarithmic asymptotic complexity, despite being implemented in a purely functional language.

Compared to other languages, a constraint imposed by Coq is that all programs must be terminating. The UFchecker (see Section 9.2.3) is trivially terminating. Termination of the proof-forest verifier is more intricate because the Explain algorithm does not terminate if the proof forest is ill-formed *e.g.,* has cycles. However, if the proof forest is well-formed, an edge

is only traversed once. As a result, at each recursive call, our verifier decrements an integer initialised to the size of the proof forest. An interesting observation is that the original Explain algorithm [NO05, Section 3.4] always terminates but does not detect certain ill-formed proof forests *e.g.,* $a \xrightarrow[a=f(a)]{b=f(b)} b$. In this case, the recursive Explain algorithm of Listings 9.2 and 9.1 does not terminate. In Coq, the verifier fails after exhausting the maximum number of allowed recursive calls.

For the sake of comparison, we have also implemented the UF proof format of Z3. Z3 refutations are also generated using Explain [dMB08b, Section 3.4.2]. Unlike ours verifiers, Z3 proofs are using explicit propositional reasoning and *modus ponens*. As a consequence formulae do not have a constant size. As already noticed by others [BW10, AFG+11], efficient verifiers require a careful handling of sharing. Our terms and formulae are *hash-consed*; sharing is therefore maximum and comparison of terms or formulae is a constant-time operation.

9.3.2 Benchmarks

We have assessed the efficiency of our UF verifiers on several families of handcrafted conjunctive UF benchmarks. The benchmarks are large and all the literals are necessary to prove non-satisfiability. For all our benchmarks the running time of the verifiers is negligible especially compared to the time spent parsing/type-checking the textual representation of the different UF proofs. Moreover, the proof size is linear in the size of the formulae. The veriT UF small checker uses a proof format similar to the command format, therefore we did not include our own implementation of the command verifier in the experiment.

Figure 9.3 on the following page shows our experimental results for a family of formulae of the general form

$$\begin{array}{l} x_0 = x_1 \qquad x_0 \neq x_{(j+1)\cdot j} \\ f(x_{i\cdot j}, x_{i\cdot j}) = x_{i\cdot j+1} = ... = x_{i\cdot j+j} \quad \text{for} \quad i \in \{0 \ldots j\} \end{array}$$

The benchmarks are indexed by the number of UF variables and the results are obtained using a Linux laptop with a processor Intel Core 2 Duo CPU T9600 (2.80GHz) and 4GB of Ram. Figure 9.3.a, on the left, shows the time needed to construct and compile Coq proof terms. Figure 9.3.b, on the right, shows the size of the compiled proof terms. Figure 9.3.c, at the bottom, is focusing on the running time of the verifiers excluding the parsing/typing phase.

For all our benchmarks, the UFchecker shows a noticeable advantage over the other verifiers. We can also remark that its behaviour is more predicable. The veriT verifier [AFG+11] is using proofs almost as small as proof forests. This is an impressive result knowing that veriT produces sometimes traces

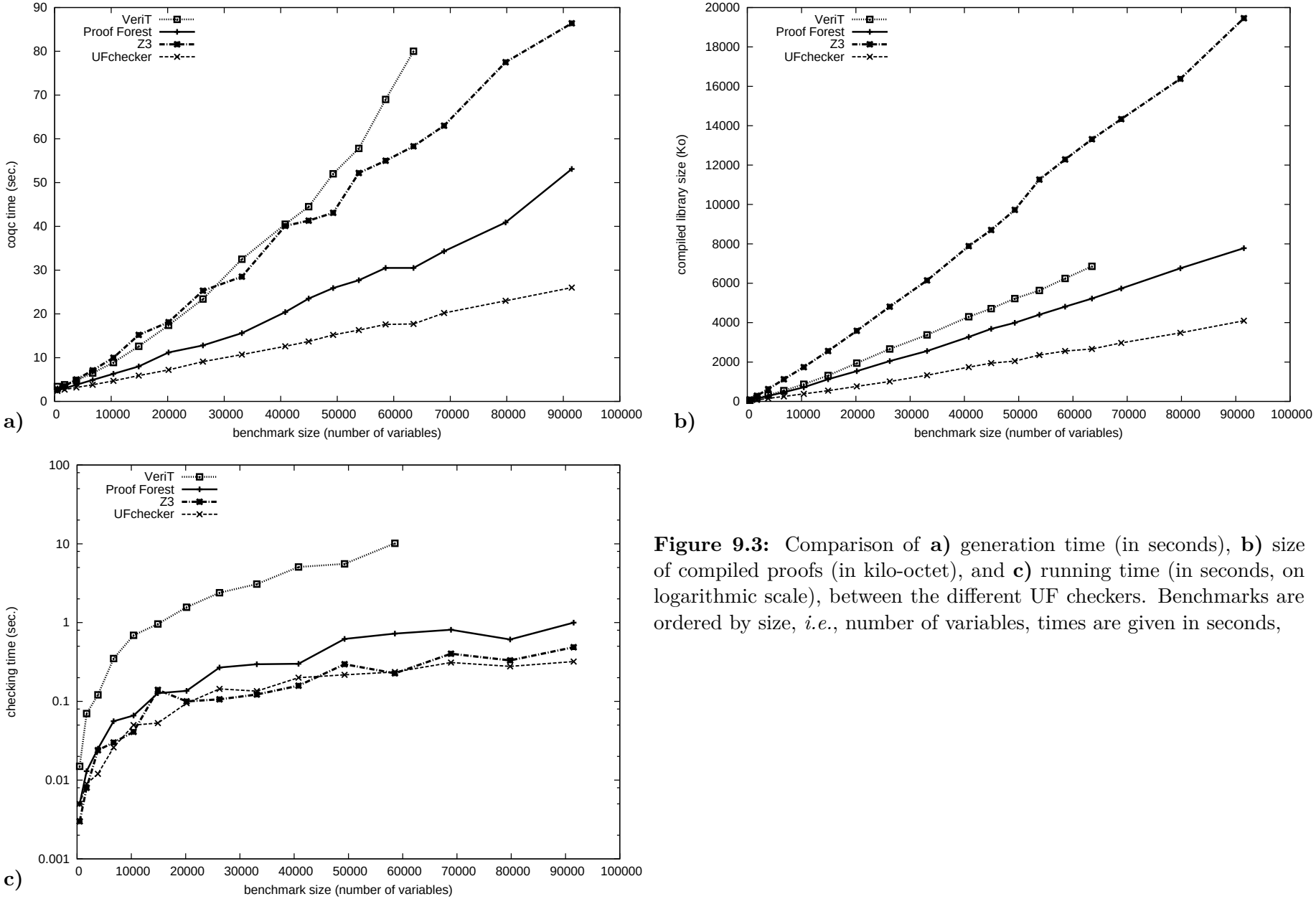


Figure 9.3: Comparison of **a)** generation time (in seconds), **b)** size of compiled proofs (in kilo-octet), and **c)** running time (in seconds, on logarithmic scale), between the different UF checkers. Benchmarks are ordered by size, *i.e.*, number of variables, times are given in seconds,

(not shown) that can be more than two orders of magnitude bigger. In the timings, the pre-processing needed to perform the proof reduction is accounted for and might explain why the VERIT verifier gets slower as the benchmark size grows. Remark also that for the biggest benchmarks, the VERIT SMT solver fails to generate proofs.

Except for VERIT, the running time of the different verifiers (Figure at the bottom) is below the second and is not the limiting factor of the verification. The Z3 verifier is more scalable despite being slightly but constantly overrun by the our verifiers. Not surprisingly, the `UFchecker` requires smaller proofs and therefore its global checking time is also smaller. For big benchmarks, even its running time tends to be smaller than other verifiers.

As all the verifiers are equally optimised, we are confident that the verifier for trimmed proof forests `UFchecker` requires smaller UF proofs in general and that it should scale better.

9.4 Conclusion

We have compared empirically different proof verifiers for UF proofs that can be generated by SMT solvers. The conclusion of our experiments is that our `UFchecker` verifier outperforms existing verifiers for UF proofs. Its certificates can be generated by the SMT solver with little overhead, the proof is succinct and the proof checking is fast. Moreover, we also have proved its soundness in COQ with respect to UF axioms. As future work, we intend to integrate the `UFchecker` into the SMT proof verifier presented in Chapter 8, and extend its scope to the logic of constructors. The experiments presented in this chapter illustrate how important the differences between different verification algorithms for a single theory can be. Conversely, they stress how useful it can be for a multi-theory result verifier to be able to update easily the algorithm it uses for a particular theory, and therefore emphasise the relevance of the modular proof-system and result verifier presented in Chapter 8.

Chapter 10

Conclusion

In this dissertation we have examined how to apply the result certification methodology to static analysers. Our approach is to divide the verification of static analysis results into two steps: first, the soundness and the precision—*i.e.*, the fact that the abstraction can prove the absence of errors—of the result of the analyser are reduced to the validity of a set of Verification Conditions (VCs); then, these verification conditions are discharged by Automated Theorem Provers (ATPs), that can themselves be certified using the result certification methodology. This approach brings modularity to the result verifier, as improving the reliability of each step improves the reliability of the whole scheme.

> Off-the-shelf automated theorem provers can be used to build efficient and trustworthy static analysis result verifiers, because the many-sorted first-order logic provides a formalism general enough to describe abstract domains in a simple enough way for decision procedures to be able to conclude.

Splitting the result certification process in successive steps multiply the number of *links* in the certification *chain*, and requires careful examination of each step, to understand which links may be trusted, and which links should not be. However, it also reduces the trusted computing base to powerful, yet trustworthy, tools, such as proof assistants, by distributing the proof effort over simpler isolated problems. Conversely, it allows the approach to benefit immediately from new solutions that may emerge for any particular sub-problem. Therefore, by splitting our initial problem into sub-problems, and by using standard tool belonging to active areas of research, we hope to secure the long term relevance of our work.

VC generation and soundness. The verification condition calculus we propose is based on standard deductive verification techniques and tools, such as defining a memory model in an Intermediate Verification Language

(IVL), and using the weakest precondition calculus of the IVL to generate the verification condition. Supposing that static analysers are formalised as abstract interpretation allows us to avoid some difficulties of deductive verification, such as the need for framing conditions. To describe the memory model, we define a theory of semantic states in the IVL, therefore the verification conditions manipulate states of execution, exactly as an operational semantics does. This allows us to prove the soundness of the verification calculus *w-r-t* the operational semantics, defined as an interpreter in the IVL. To do this proof, we use standard deductive verification techniques but we do not rely on the translation of programs into the IVL, thus we exclude from the Trusted Computing Base (TCB) any form of compilation. The generated VCs may be more detailed than VCs obtained in deductive verification, but can nonetheless be discharged automatically by ATPs in a matter of seconds for numerical analyses.

> To obtain an executable result verifier and prove its soundness we only need to specify the formalisation of the static analysis as abstract interpretation and the operational semantics of the language in a intermediate verification language.

The VCs generated to certify the results of analyses of the heap are too complex for ATPs. To resolve this difficulty, we proposed another simpler VC calculus, that produces *quantifier free formulae*. To obtain such simple conditions, the new VC calculus is necessarily less general, nonetheless, it can be used for a *family* of analysers, defined by a parametrised abstract domain and concretisation. The use of an IVL to generate VCs, and prove the soundness of the VC calculus, gives a framework to prove the soundness of the quantifier free VC calculus *w-r-t* the general VC calculus. This layered, modular design allows us to split the proof effort between ATPs and a proof assistant, when possible—*i.e.*, when the proof need not be redone for individual results. We also benefit from the back-end of the IVL, which allows us to use different ATPs, which has proved invaluable when discharging the VCs.

Testing static analysers. The modularity of the approach, the use of an IVL to specify the VC calculus, and the ability to rely on off-the-shelf ATPs to discharge VCs, mean that the result verifier can be operable with very little programming.This simplifies substantially another application of the result certification methodology: the use of the result verifier as an oracle to test the analyser during its development. The definition of the VC calculus is based on the formalisation of the analyser in the abstract interpretation framework, thus a new result verifier can be obtained by simply declaring the abstract domains and concretisation function in the IVL. As soon as the analyser can output abstractions, the IVL can be used to generate verification conditions that can be discharged using off-the-shelf ATPs. Therefore,

testing if an analyser can prove the absence of errors for a given program can be done automatically, and testing if a modification in the implementation of the analyser generates unsound results only requires running the new version of the analyser and embedding the results in the IVL.

SMT result certification. To prove the validity of verification conditions, we use Automated Theorem Provers to prove the *unsatisfiability* of the negation of many-sorted first order formulae. Therefore to complete our result certification approach for static analysis, we have applied the result certification methodology to a specific family of automated theorem provers: Satisfiability Modulo Theory (SMT) solvers. The certification of SMT solvers is a first step towards the certification of ATPs. Furthermore, SMT solvers are commonly used in a wide variety of program verification techniques, therefore verifying their results have far more application than our particular methodology for static analysers. Conversely, by splitting our initial problem into sub-tasks, we benefit from future improvements in ATP certification.

Our experiments emphasised the need for different SMT solvers to discharge the verification conditions, but off-the-shelf proof-producing solvers currently do not agree on a proof-format, and their different behaviour stems from different design choices. To be able to use the same result verifier for different tools, we have developed a general proof-system using the mechanisms common to all SMT solvers rather than concentrating on the output of a particular solver. The proof-system uses *conflict-clauses* to separate the propositional reasoning from the theory reasoning, which can then be certified in their own proof-system. This allows us to use the standard API of existing solvers to emulate a proof-producing SMT solver that outputs conflict-clauses, even though off-the-shelf solvers currently does not provide this information, and to certify these clauses using a much simpler SMT implementation that can output certificates in a format or our choice. Our experiments show that it is sufficient to discharge a substantial subset of the industrial benchmarks used by the SMT community to evaluate the progresses in SMT solving algorithms. Moreover, our experiments provide strong evidences that the limiting factor to discharge even larger proofs is neither the certification of conflict-clauses nor the efficiency of the verifier's algorithm, but the manipulation of large terms by the Coq engine which we used to execute the verifier. As the result verifier can be extracted to OCaml, an efficient programming language, this limit can be easily overcome.

> To be certified efficiently, SMT solvers only need to output conflict-clauses.

A modular result verifier. Moreover, to leave room for additional theory combinations, we designed a modular result verifier, based on a combi-

nation of mono-theory sub-verifiers. By doing so, we were able to describe combinations of theories without committing ourselves to a specific certificate format, nor requiring to encode certificates in a particular higher-order logic. The benefit of this modular design was illustrated on the theory of uninterpreted function, for which we defined and compared several certificates format and verification algorithms, with different properties regarding the size of the certificates, the complexity of the verifier, and its efficiency. By implementing the result verifier in the language of the proof assistant Coq, based on dependent types, the modularity of the implementation of the result verifier can be lifted to the proof of its correctness, and the proof of the general architecture does not need to be changed when new sub-verifiers are added.

Improving the partial automation of a proof assistant. Using the proof-by-reflection technique, the result verifier can be used to integrate SMT solvers in Coq, allowing to call these powerful decision procedures in the middle of a proof script. This improves the automation of the proof-assistant, in addition to improving the reliability of SMT solvers. As the proof-by-reflection requires to execute the verifier in the Coq engine, the limit *w-r-t* the size of proof-terms exhibited in the experiment apply, however, using SMT solvers to provide *partial* automation inside proof scripts does entails neither large formulae nor large proofs. Therefore, the result verifier could provide much needed automation before scalability issues arise.

Further work. Besides further engineering work needed to bring together all pieces of software, there are still open questions to answer to complete the scheme. We have proposed a result certification scheme for SMT solvers but not for TPTP solvers, which are also needed to discharge some conditions in our experiments. Furthermore, the result certification of SMT solvers is not complete enough at the moment, as it lacks certificates for axiom instantiation and for some theories needed to describe the theory of semantic states. Moreover, the encoding of the theory of semantic states could be more direct, and new decision procedures for additional theories of Many-Sorted First-Order Logic would be useful to discharge more efficiently, and more consistently, the VCs. How to isolate parts of the theory of semantic states to benefit from its structure beyond what can be achieved with the theory of maps and algebraic data-types is an open question. In addition, more information could be extracted from the result of the ATPs when failing to verify the results of an analyser: the model of the negation of a VC can be translated into a counter-example, *i.e.*, values of the program on which it violates the invariant deduced from the abstraction returned by the analyser.

The question of finding a *systematic* process to generate new verification

condition calculus that produce VCs that automated provers can easily discharge constitute a more ambitious research project. If a family of analyses is described by a parametrisation of the abstract domain and of the concretisation, it seems that an equivalent of the VC calculus $\mathcal{DVC}_{obj}$ could be obtained by partial evaluation of the pre-condition and post-condition predicates on the parametrised concretisation, and a simplification of the general verification conditions on the partially evaluated predicates. The simplification step, which is probably the most important part, may be guided by invariants of the semantics of the language and by the transfer function. Another possibility may be to find a way to extract the VC calculus from the proof of soundness of the formalisation of the analyser. In this case, the benefit of the result certification approach comes from the fact that the actual implementation of the analyser may be quite different from its formalisation, in order to be efficient, *e.g.*, the proof of the formalisation may use a naive fix-point engine, a high-level description of the abstract domains, and the termination does not need to be proved. Such an approach would mean that once the fundamental ideas behind an analysis are proved sound, the results of the analyser can be fully trusted—*i.e.*, automatically verified—regardless of the details of its implementation, filling the gap between formal verification techniques and efficient programming a little more.

Chapter 11

Bibliography

[AAV02] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS'02*, pages 75–86, Copenhagen, Denmark, July 2002. IEEE Computer Society.

[ABB+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís M. Pereira, editors, *Proceedings of the European Workshop on Logics in Artificial Intelligence, JELIA'00*, volume 1919 of *Lecture Notes in Computer Science*, pages 21–36, London, UK, 2000. Springer.

[ABC+02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction, CADE'02*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210, Copenhagen, Denmark, July 2002. Springer.

[ABG+11] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. Verified resource guarantees using COSTA and KeY. In Siau-Cheng Khoo and Jeremy Siek, editors, *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation, PEPM '11*, SIGPLAN, pages 73–76, New York, NY, USA, January 2011. ACM.

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In Susanne Biundo and Maria Fox, editors, *Recent Advances in AI Planning, Proceedings of the 5th European Conference on Planning, ECP'99*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108, Durham, UK, September 2000. Springer.

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Mark N. Wegman and Thomas W. Reps, editors, *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'00*, pages 243–253, Boston, Massachusetts, USA, January 2000. ACM.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Proceedings of the First International Conference on Certified Programs and Proofs, CPP'11*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150, Kenting, Taiwan, December 2011. Springer.

[AGH+05] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–26, Marseille, France, March 2005. Springer.

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In Matt Kaufman and Lawrence C. Paulson, editors, *Proceedings of the First International Conference on Interactive Theorem Proving, ITP'10*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98, Edinburgh, UK, July 2010. Springer.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 23(5):657–683, September 2001.

[APH05] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. Abstraction-carrying code. In Franz Baader and Andrei Voronkov, editors, *Proceedings of the 11th International Conference, Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'05*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397. Springer, March 2005.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, LICS 2001*, pages 247–256, Boston, MA, USA, June 2001. IEEE Computer Society, Washington, DC, USA.

[BB02] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, April 2002.

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In Kousha Etessami and SriramK. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification, CAV'05*, volume 3576 of *Lecture Notes in Computer Science*, pages 335–349, Edinburgh, UK, July 2005. Springer.

[BBN11] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 8th International Symposium on Frontiers of Combining Systems, FroCoS'11*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27, Saarbrücken, Germany, October 2011. Springer.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction, CADE'11*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130, Wrocław, Poland, August 2011. Springer.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 4th International Conference on Formal Methods for Components and Objects, FMCO'05*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, Amsterdam, The Netherlands, November 2005. Springer.

[BCF+06] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: A comparative analysis. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'06*, volume 4246 of *Lecture Notes in Computer Science*, pages 527–541, Phnom Penh, Cambodia, November 2006. Springer.

[BCJ12] Frédéric Besson, Pierre-Emmanuel Cornilleau, and Thomas Jensen. Why3 and Coq source of the development, 2012. available at `http://www.irisa.fr/celtique/ext/chk-sa`.

[BCP11] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Proceedings of the First International Conference on Certified Programs and Proofs, CPP'11*, volume 7086 of *Lecture Notes in Computer Science*, pages 151–166, Kenting, Taiwan, December 2011. Springer.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Proceedings of the First International Conference on Certified Programs and Proofs, CPP'11*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377, Kenting, Taiwan, December 2011. Springer.

[Bes07] F Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Proc. of TYPES 2006*, volume 4502 of *LNCS*, pages 48–62. Springer, 2007.

[BFM+12] François Bobot, Jean-christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The Why3 platform (version 0.80). Technical Report October, University Paris-Sud, CNRS, Inria, 2012.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[BFSW11] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. Reconstruction of z3's bit-vector proofs in HOL4 and Isabelle/HOL. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Proceedings of the First International Conference on Certified Programs and Proofs, CPP'11*, volume 7086 of *Lecture Notes in Computer Science*, pages 183–198, Kenting, Taiwan, December 2011. Springer.

[Bie08] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008.

[BJPT10] Frédéric Besson, Thomas Jensen, David Pichardie, and Tiphaine Turpin. Certified result checking for polyhedral analysis of bytecode programs. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Proceedings of the 5th Symposium on Trustworthly Global Computing, TGC'10*, volume 6084 of *Lecture Notes in Computer Science*, pages 253–267, München, Germany, February 2010. Springer.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Marseille, France, March 2004. Springer.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transaction on Software Engineering*, 21(10):785–798, October 1995.

[BN10] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Proceedings*

of the 5th International Joint Conference on Automated Reasoning, IJCAR'10, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121, Edinburgh, UK, July 2010. Springer.

[BODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction, CADE'09*, lecture Notes in Computer Science, pages 151–156, Montreal, Canada, August 2009. Springer.

[Bou94] Richard John Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, 1994.

[BP11] François Bobot and Andrei Paskevich. Expressing polymorphic types in a many-sorted language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 8th International Symposium on Frontiers of Combining Systems, FroCoS'11*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102, Saarbrücken, Germany, October 2011. Springer.

[BT07] Clark Barrett and Cesare Tinelli. Cvc3. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[BW10] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the First International Conference on Interactive Theorem Proving, ITP'10*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194, Edinburgh, UK, July 2010. Springer.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages, POPL'77*, pages 238–252, Los Angeles, California, USA, January 1977. ACM.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N.

Zilles, and Barry K. Rosen, editors, *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL'79*, pages 269–282, San Antonio, Texas, USA, January 1979. ACM.

[CC92a] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the 4th International Workshop Programming Language Implementation and Logic Programming, PLILP'92,*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-VerlagLeuven, Belgium, August 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In Ravi Sethi, editor, *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'92*, pages 83–94, Albuquerque, New Mexico, USA, January 1992. ACM Press.

[CC05] Evelyne Contejean and Pierre Corbineau. Reflecting proofs in first-order logic with equality. In Robert Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction, CADE'05*, volume 3632 of *Lecture Notes in Computer Science*, pages 7–22, Tallinn, Estonia, July 2005. Springer.

[CCK06] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[CCKL07] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight integration of the ergo theorem prover inside a proof assistant. In *Proceedings of the second workshop on Automated formal methods*, AFM '07, pages 55–59. ACM, 2007.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In Gregory Morrisett and Manuel Fähndrich, editors, *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, TLDI'05*, pages 1–12, Long Beach, CA, USA, January 2005. ACM.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and

Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs'09*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, August 2009. Springer.

[CF08] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In Sophia Drossopoulou, editor, *Proceedings of 17th European Symposium on Programming, ESOP'08*, volume 4960 of *Lecture Notes in Computer Science*, pages 322–336, Budapest, Hungary, March 2008. Springer.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL'78*, pages 84–96, Tucson, Arizona, USA, January 1978. ACM.

[Che68] N. V. Chernikoba. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[CJPR05] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, September 2005.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In Monica S. Lam, editor, *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI'00*, pages 95–107, Vancouver, British Columbia, Canada, June 2000. ACM.

[CMTS09] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 4th International Workshop on Systems Software Verification, SSV'09*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103, Aachen, Germany, June 2009. Elsevier Science Publishers B. V.

[CO81] Robert Cartwright and Derek C. Oppen. The logic of aliasing. *Acta Informatica*, 15(4):365–384, August 1981.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In Alex Aiken and Greg Morrisett, editors, *Conference Record of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'03*, volume 38, pages 198–212, New Orleans, Louisisana, USA, January 2003. ACM.

[DdM06] Bertrand Dutertre and Leonardo M. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Dij97] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5(7):394–397, 1962.

[dMB08a] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, May 2008.

[dMB08b] Leonardo M. de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*, volume 418, Doha, Qatar, November 2008. CEUR-WS.org.

[dMB08c] Leonardo M. de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS'08*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, Hungary, March 2008. Springer.

[dMR02] Leonardo M. de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. *. . . Theory and Applications of Satisfiability Testing*, (May):1–8, 2002.

[dMRS05] Leonardo M. de Moura, Harald Rueß, and Natarajan Shankar. Justifying equality. In *Selected Papers from the Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning, PDPAR'04*, volume 125 of *Electronic Notes in Theoretical Computer Science*, pages 69–85, Cork, Ireland, July 2005. Elsevier.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[ERB+06] Christian Engel, Andreas Roth, Abian Blome, Richard Bubel, and Simon Greiner. KeY quicktour for JML. pages 1–17, 2006. available at `http://www.key-project.org/case_studies`.

[FG02] Pascal Fontaine and E. Pascal Gribomont. Using BDDs with combinations of theories. In Matthias Baaz and Andrei Voronkov, editors, *Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'02*, volume 2514 of *Lecture Notes in Computer Science*, pages 190–201, Tbilisi, Georgia, October 2002. Springer.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In Warren A. Hunt, Jr and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification, CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367, Boulder, CO, USA, July 2003. Springer.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. *Proceedings of the 18th ACM SIGPLAN International Conference on Object Oriented Programing, Systems, Languages, and Applications, OOPSLA'03*, pages 302–312, October 2003.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI'02*, pages 234–245, Berlin, Germany, June 2002. ACM.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 1967.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'98*, pages 310–327, Vancouver, British Columbia, Canada, October 1998. ACM.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[FMM+06] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor P. Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS'06*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181, Vienna, Austria, March 2006. Springer.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In Chris Hankin and Dave Schmidt, editors, *Conference Record of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'01*, pages 193—-205, London, UK, January 2001. ACM.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188, Boston, Massachusetts, USA, 2004. Springer.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proc. of ICFP 2002*, pages 235–246, 2002.

[GM05] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Oxford, UK, August 2005. Springer.

[GN03] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the conference on Design, Automation and Test in Europe, DATE'03*, volume 1, pages 10886—-10891, Munich, Germany, March 2003. IEEE Computer Society.

[GN07] Evguenii I. Goldberg and Yakov Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, June 2007.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In Samson Abramsky and T. S. E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192, Brighton, UK, April 1991. Springer.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a hoare logic with abrupt termination. In T. S. E. Maibaum, editor, *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering, FASE'00, held as part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303, Berlin, Germany, March 2000. Springer.

[HJP08] Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In Gilles Barthe and Frank S. de Boer, editors, *Proceedings of the 10th IFIP WG 6.1 International Conference Formal Methods*

for Open Object-Based Distributed Systems, FMOODS'08, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149, Oslo, Norway, June 2008. Springer.

[HMM12] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Proceedings of the 4th international conference on Verified Software: Theories, Tools, Experiments, VSTTE'12*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17, Philadelphia, PA, USA, 2012. Springer.

[Hoa69] C. a. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[HST+03] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3-4):191–229, November 2003.

[Hur99] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–322, Nice, France, September 1999. Springer.

[Jea] Bertrand Jeannet. The Interproc analyzer.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In Kazunori Ueda, editor, *Proceedings of the 8th Asian conference on Programming languages and systems, APLAS'10*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311, Shanghai, China, December 2010. Springer.

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Proceedings of the Third international conference on NASA Formal methods, NFM'11*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55, Pasadena, CA, USA, April 2011. Springer.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of the 14th international symposium of Formal Methods, FM'06*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, August 2006. Springer.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2003.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML : A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[Lei10] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR'10*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370, Dakar, Senegal, April 2010. Springer.

[Ler03] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of POPL 2006*, pages 42–54. ACM, 2006.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[MBG06] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and cvc lite. *ENTCS*, 144(2):43–51, 2006.

[MF09] Claude Marché and Jean-Christophe Filliâtre. The Krakatoa tool for deductive verification of Java programs, 2009.

[Min04] Antoine Miné. *Weakly relational numerical abstract domains.* Phd thesis, École Polytechnique, Palaiseau, France, December 2004.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, March 2006.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference, DAC'01*, pages 530–535, Las Vegas, Nevada, USA, June 2001. ACM.

[Moo62] Ramon E. Moore. *Interval arithmetic and automatic error analysis in digital computing.* Phd dissertation, Stanford University, Stanford, CA, USA, November 1962.

[Mos08] Michał Moskal. Rocket-fast proof checking for SMT solvers. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS'08*, volume 4963 of *Lecture Notes in Computer Science*, pages 486–500. Springer, March 2008.

[MPm05] Claude March and Christine Paulin-mohring. Reasoning about Java programs with aliasing and frame conditions. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194, Oxford, UK, August 2005. Springer.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, January 2004.

[MT93] Karl Meinke and John V Tucker. *Many-sorted Logic and its Applications.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 21(3):527–568, May 1999.

[Nec97] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'97*, pages 106–119, Paris, France, January 1997. ACM.

[Nec98] George C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, 1998.

[NL98a] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI'98*, pages 333–344, Montreal, Canada, 1998. ACM.

[NL98b] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, LICS'98*, pages 93–104, Indianapolis, Indiana, USA, June 1998. IEEE Computer Society.

[NL98c] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91. Springer, 1998.

[NL00] George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In David A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction, CADE'00*, volume 1831 of *Lecture Notes in Computer Science*, pages 25–44, Pittsburgh, PA, USA, June 2000. Springer.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 1999.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 1(2):245–257, October 1979.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications, Proceedings of the 16th International Conference on Rewriting Techniques and Applications, RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468, Nara, Japan, April 2005. Springer.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, April 2007.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(t). *Journal of the ACM*, 53(6):937–977, November 2006.

[NS03] George C. Necula and Robert R. Schneck. A sound framework for untrusted verification-condition generators. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science, LICS'03.*, pages 248–260, Ottawa, Canada, June 2003. IEEE Computer Society.

[Pas09] Andrei Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform. Research Report RR-7128, INRIA, 2009. available at `http://hal.inria.fr/inria-00439232`.

[Pic05] David Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés.* PhD thesis, Université Rennes∼1, 2005. in French.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the16th International Conference on Automated Deduction, CADE'99*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer.

[PS07] Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'07*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245, Kaiserslautern, Germany, September 2007. Springer.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 1999.

[Rus81] John M. Rushby. Design and verification of secure systems. In John Howard and David P. Reed, editors, *Proceedings of the eighth ACM symposium on Operating systems principles, SOSP'81*, number 5, pages 12–21, New York, NY, USA, December 1981. ACM.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction, CADE'99*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296, Trento, Italy, July 1999. Springer.

[SB06] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining BDDs. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *First International Computer Science Symposium in Russia, Theory and Applications, CSR'06*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611, St. Petersburg, Russia, June 2006. Springer.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[Sch02] Stephan Schulz. E – a brainiac theorem prover. *AI Communication*, 15(2-3):111–126, August 2002.

[STB10] Aaron Stump, Cesare Tinelli, and Clark Barrett. The SMT-LIB standard: Version 2.0, 2010.

[Ste73] Gilbert Stengle. A nullstellensatz and a positivstellensatz in semialgebraic geometry. *Mathematische Annalen*, 207(2):87–97, 1973.

[Stu09] Aaron Stump. Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 228:121–133, January 2009.

[Sut09] Geoff Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, July 2009.

[Van02] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Online proceedings of the 7th International Symposium on Artificial Intelligence and Mathematics, AMAI'02*, pages 1–10, Fort Lauderdale, Florida, USA, January 2002.

[Van07] Allen Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In João Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 10th International Conference theory and Applications of Satisfiability Testing, SAT'07*, Lecture Notes in Computer Science, pages 328–333, Lisbon, Portugal, May 2007. Springer.

[Van12] Allen Van Gelder. Producing and verifying extremely large propositional refutations. *Annals of Mathematics and Artificial Intelligence*, 65(4):329–372, November 2012.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Tools*

and Algorithms for the Construction and Analysis of Systems, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[VJP10] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC'10*, pages 2517–2522, Sierre, Switzerland, March 2010. ACM.

[WA09] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic, Special Issue: Empirically Successful Computerized Reasoning*, 7(1):26–40, March 2009.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[Weg74] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–113, February 1974.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[Wil85] Maurice Wilkes. *Memoirs of a Computer Pioneer*. MIT Press, 1985.

[WN05] Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. In Mooly Sagiv, editor, *Proceedings of the 14th European Symposium on Programming Languages and Systems, ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 326–341, Edinburgh, UK, April 2005. Springer.

[WNKN04] Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping proof carrying code. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science, TCS'04*, volume 155 of *IFIP International Federation for Information Processing*, pages 333–347, Toulouse, France, August 2004. Springer.

[ZG03] Hans Zantema and Jan Friso Groote. Transforming equality logic to propositional logic. *Electronic Notes in Theoretical Computer Science*, 86(1):162–173, 2003.

[Zha03] Lintao Zhang. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. of DATE 2003*, pages 10880–10885, 2003.

Appendix A

Certificate checking and generation for LRA and LIA

In this section we introduce the certificate language and proof system for linear arithmetic and describe its certifying prover. Literals are of the form $e \bowtie 0$ with e a linear expression manipulated in (Horner) normal form and $\bowtie \in \{\geq, >, =\}$.

Certificate language. Since our initial work [Bes07], we are maintaining and enhancing reflexive tactics for real arithmetic (`psatz`) and linear integer arithmetic (`lia`). Those tactics, which are now part of the Coq code-base, are based on the *Positivstellensatz* [Ste73], a rich proof system which is complete for non-linear (real) polynomial arithmetic. Those reflexive verifiers are at the core of our current theory verifiers for linear real arithmetic (LRA) and linear integer arithmetic (LIA). We present here simplified proof systems specialised for linear arithmetic.

For linear real arithmetic Farkas' lemma provides a sound and complete notion of certificate for proving that a conjunction of linear constraints is unsatisfiable [Sch98, Corollary 7.1e]. It consists in exhibiting a positive linear combination of the hypotheses that is obviously unsatisfiable, *i.e.*, deriving $c \bowtie 0$ for $\bowtie \in \{>, \geq, =\}$ and c a constant such that $c \bowtie 0$ does not hold. To construct such a contradiction, we start with a sub-proof system that allows to derive an inequality with a list of commands (a Farkas certificate). Each command is a pair $\texttt{Mul}(c, i)$ where c is a coefficient (in type $\mathbb{Z}$) and i the index of an assumption in the current assumption set. Such a command is used below in a judgement $\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma', e' \bowtie' 0$ with $\bowtie$ and $\bowtie'$ in $\{\geq, >\}$. $\Gamma \cup \{e \bowtie 0\}$ is the current set of assumptions, $e' \bowtie' 0$ is a new deduced inequality and Γ' is an enriched set of assumptions. For LIA, the proof system is augmented with a `Cut` command to generate *cutting planes* [Sch98, chapter 23] and a rule for case-splitting `Enum`. We also need a `Push` and a `Get` command in order to update the environment and

$$\frac{c > 0 \quad \Gamma(i) \mapsto e' \geq 0}{\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma, (c[*]e'[+]e) \bowtie 0} \qquad \frac{\Gamma(i) \mapsto e' = 0}{\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma, (c[*]e'[+]e) \bowtie 0}$$

$$\frac{c > 0 \quad \Gamma(i) \mapsto e' > 0}{\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma, (c[*]e'[+]e) > 0} \qquad \frac{\Gamma(i) = e' \bowtie 0}{\Gamma, e \bowtie' 0 \xrightarrow{Get(i)} \Gamma, e' \bowtie 0}$$

$$\frac{\Gamma' = \Gamma[i \mapsto e \bowtie 0]}{\Gamma, e \bowtie 0 \xrightarrow{Push(i)} \Gamma', e \bowtie 0} \qquad \frac{g > 0}{\Gamma, (g[*]e[-]d) \geq 0 \xrightarrow{\texttt{Cut}} \Gamma, (e[-]\lceil d/g \rceil) \geq 0}$$

$$\frac{g \mid d}{\Gamma, (g[*]e[-]d) = 0 \xrightarrow{\texttt{Cut}} \Gamma, (e[-](d/g)) = 0} \qquad \frac{\neg(g \mid d)}{\Gamma, (g[*]e[-]d) = 0 \xrightarrow{\texttt{Cut}} \Gamma, 0 > 0}$$

$$\frac{\begin{array}{c}\Gamma(i_1) \mapsto e[-]l \geq 0 \quad \Gamma(i_2) \mapsto h[-]e \geq 0 \\ \forall v \in [l, h], \Gamma, e = v \xrightarrow{c_{v-l}}_* \Gamma'_v, e' \bowtie' 0\end{array}}{\Gamma, \cdot \bowtie 0 \xrightarrow{\texttt{Enum}(i_1, i_2, [c_0; \ldots; c_{h-l}])} \Gamma, e' \bowtie' 0}$$

Figure A.1: LRA and LIA proof rules

retrieve an already derived formula. The semantics of the commands is given in Figure A.1. The operators $[*], [+], [-]$ model the standard arithmetic operations but maintain the normalised form of the linear expressions. The rules for the `Mul` command follow the standard sign rules in arithmetic: for example, if e' is positive we can add it c times to the right part of the inequality $e \bowtie 0$, assuming c is strictly positive. To implement the `Cut` rule, the constant g is obtained by computing the greatest common divisor of the coefficient of the linear expression. For inequalities, the rule allows to *cut* the constant. For equalities, it allows to detect a contradiction if g does not divide d ($\neg(g \mid d)$).

A LRA certificate is then either a proof of $0 > 0$ given by a list of commands or a proof of $x = y$ given by two lists of commands (one for $x - y \geq 0$ and one other for $y - x \geq 0$.

```
Inductive LRA_certificate :=
|LRA_False (l : list command) |LRA_Eq (l1 l2 : list command)
```

$$\frac{\Gamma \vdash l : 0 > 0}{\Gamma \vdash_{LRA} (\texttt{LRA_False}(l)) : (\Gamma, nil)} \qquad \frac{\Gamma \vdash l_1 : e \geq 0 \quad e = x[-]y \quad \Gamma \vdash l_2 : [-]e \geq 0}{\Gamma \vdash_{LRA} (\texttt{LRA_Eq}(l_1, l_2)) : (\Gamma, [x = y])}$$

Because the theory LIA is non-convex, it is necessary to deduce contradictions but also disjunction of equalities.

```
Inductive LIA_certificate :=
| LIA_False (l : list command)
```

```
| LIA_Eq (eqs : list (var * var)) (l : list (list command))
```

Proving equalities is done by performing a case-split and each list of commands $l \in l$ is used to prove that a case is unsatisfiable.

Certificate generation. In order to produce Farkas certificates efficiently, we have implemented the Simplex algorithm used in Simplify [DNS05]. This variant of the standard linear programming algorithm does not require all the variable to be non-negative, and directly handles (strict and large) inequalities and equalities. Each time a contradiction is found, one line of the Simplex tableau gives us the expected Farkas coefficients. The algorithm is also able to discover new equalities between variables. In this case again, the two expected Farkas certificates are read from the current tableau, up to trivial manipulations.

For LIA, we use a variant of the Omega test [**?**]. The Omega test lacks a way to derive equalities but the number of shared variables is sufficiently small to allow an exhaustive search. Moreover, an effective heuristics is to pick as potential equalities the dis-equalities present in the unsat core.

List of Listings

2.1 Example of a program that requires relational information . . 20
2.2 Binary search 23
2.3 Example of a constructor annotated with raw types 34
5.1 Implementation of the interpreter's annotations 77
5.2 Lemma specifying the VC calculus 77
5.3 Example of an instantiation with intermediary lemma 78
5.4 Pattern used for the interpreters 79
5.5 Transitive closure of the eval function 80
5.6 Well-formedness of programs 81
5.7 Functions manipulating arrays 82
6.1 Specification of $\mathcal{DVC}$ 104
8.1 Environment API definition 126
8.2 Theory record definition 126
9.1 ExplainAlongPath 139
9.2 Recursive Explain algorithm 139
9.3 Verifier algorithm for trimmed forests 141

List of Figures

2.1 Syntax of numerical programs. 14
2.2 Natural semantics of expressions and tests 17
2.3 Semantics of instructions 17
2.4 Error conditions 18
2.5 Syntax of object-oriented programs. 25
2.6 Semantics of the object-oriented core language 29
2.7 Limited subset of the error conditions 29
2.8 Non-null types lattice 33

3.1 Flowchart representation of test and assignment 36
3.2 Simple memory model 39
3.3 Overview of deductive verification based on IVLs 42

4.1 Comp. between deduc. verif. and result certif. 50
4.2 Theories of semantic states 55
4.3 Translation of the natural semantics of expressions 61
4.4 VC function for assignments 62
4.5 VC functions for tests 62
4.6 VC functions for procedure calls 64
4.7 VC functions for array statements 64
4.8 VC functions for the Getfield and GetfieldNullP rules 67
4.9 VC functions for the Putfield and PutfieldNullP rules 67
4.10 Definition of the initial state of a method call 67
4.11 VC functions for Call rule 69
4.12 Global VCs 69

5.1 Overview of the implementation 76
5.2 Abstract definition of flowchart 76

6.1 Differences between inst. and non-inst. semantics 91
6.2 Short-hands used in the simplified VCs 97
6.3 VCs proving the absence of errors 101
6.4 VCs proving the soundness of the result of an analysis 102

7.1 Running example of a many sorted formula 110
7.2 Summary of the UF theory 110
7.3 Example of setup of a lazy SMT scheme 111
7.4 Example of Lazy SMT run 112
7.5 Example of Nelson-Oppen equality exchange 113

8.1 Proof rule merging propositional and theory reasoning 121
8.2 Example of an SMT certificate 122
8.3 Proof rule modelling a Nelson-Oppen equality exchange . . . 123
8.4 Example of Nelson-Oppen certificate seen as a tree 123
8.5 Generation of conflict clauses 128
8.6 Number of conflic-clauses per formula 131
8.7 Average size of the conflict-clauses 132

9.1 Example of proof forest with two trees 138
9.2 Example of trimmed forest 141
9.3 Comparison between U.F. verifiers 143

A.1 LRA and LIA proof rules 171

List of Tables

2.1 Shorthands used in the semantics of the core num. language . 16
2.2 Shorthands used in the semantics of the core OO language . . 27

5.1 Result for $VC^{pre}_{\texttt{Call}}$ on polyhedral analysis of factorial program 84
5.2 Result for $VC^{post}_{\texttt{Call}}$ on polyhedral analysis of factorial program 84

8.1 Experimental results for selected SMT-LIB logics 130