

UNIVERSITÉ PARIS 7 – DENIS DIDEROT

UFR d'INFORMATIQUE

Couloir 56-46 - 1er étage; 2, place Jussieu - 75251 Paris Cedex 05

THÈSE

pour l'obtention du Diplôme de

**DOCTEUR DE L'UNIVERSITÉ PARIS VII
SPÉCIALITÉ : INFORMATIQUE**

présentée et soutenue publiquement

par

Eva ROSE

le 27 septembre 2002

Titre :

**Vérification de Code d'Octet de la Machine Virtuelle Java
Formalisation et Implantation**

Directeur de Thèse :

Jean GOUBAULT-LARRECQ

JURY

MM.	Xavier Véronique Viguié	LEROY DONZEAU-GOUGE	Rapporteurs
	Vincent Sophia Thomas	DANOS DROSSOPOULOU JENSEN	Examineurs

Resumé

L'environnement d'exécution Java est idéal lorsqu'il s'agit de charger le code mobile binaire Java par l'Internet de façon fiable. Les classes Java sont chargées par le chargeur des classes Java ("class loader") avant d'être soumises à une vérification de conformité de typage du code binaire. Le vérificateur ("verifier") standard (spécifié par Lindholm et Yellin) réalise une stratégie de flôt de données ("dataflow algorithm"). Il est cependant difficile pour un vérificateur standard de tourner sur un système restreint en mémoire, vu que cette technique en général consomme de la mémoire RAM ("random access memory"), proportionnelle à la largeur du code.

Dans cette thèse, nous proposons d'appliquer la stratégie générale de "Proof Carrying Code" (PCC), afin de pouvoir définir une technique alternative qui permet de vérifier la conformité des typage du code binaire sur les systèmes restreints en mémoire. En effet, cette technique permet d'effectuer la vérification sans appliquer de méthodes chryptographiques, et donc sans dépendre sur d'un tiers. La technique, nommée *lightweight bytecode verification*, suggère de munir le code mobile binaire Java avec des "certificats", ce qui permet de réduire la consommation générale de mémoire. Nous définissons cette technique en utilisant une spécification formelle, LBV, et une spécification formelle du vérificateur standard, BV, ainsi qu'une spécification formelle d'un composant Java qui produit des certificats, LBC. Nous montrerons que les garanties de conformité de typage posées par LBV et BV (pour le même code) sont équivalents pour un sous-ensemble pertinent de la machine virtuelle Java. Par conséquent, la conformité de typage du code binaire ne peut être compromis ni par un faux certificat, ni par une modification directe du code. Nous dirons que LBV est "sûre" ("tamper proof").

Nous montrerons également que le vérificateur de la machine virtuelle KVM de l'entreprise Sun peut être simulé par notre technique. Ceci vaut également pour le "on-card" vérificateur, conçu par Xavier Leroy pour le code binaire destiné aux Java Cards.

Enfin, nous présentons une réalisation en Java d'un prototype de LBV pour les classes Java.

Mots clés : code d'octet virtuel Java, conformité de typage, vérification statique, machine virtuelle Java (JVM), code mobile, sémantique formelle, Internet, certificat, systèmes restreints, KVM, Java Card.

Abstract

The Java runtime environment provides for mobile code: classes can be loaded in compiled form over the Internet, as *bytecode*, and are verified for *type safety* to ensure that they can be safely loaded dynamically into the running program. Type safety is ensured by the *bytecode verifier*, a component of the class loader, and thus a security-critical component to the Java model. Sun's standard bytecode verifier implements a data-flow analysis in the form of a fixed point iteration over all possible execution paths in the method bytecode, which is not feasible on small execution platforms, as it consumes an amount of random access memory proportional to the size of the code.

In this thesis, we propose how to take a Proof-Carrying Code (PCC) approach to bytecode verification, to define an alternative bytecode verification technique called *lightweight bytecode verification*, which allows verification to be feasible on resource-constrained Java execution platforms without depending on cryptographic technology that require having to trust an external party. Instead the bytecode is annotated with a "certificate" that permits bytecode verification to proceed with much lower resource consumption. We define our technique formally by giving three formal systems for a substantial subset of the Java Virtual Machine: the LBV system defines the lightweight bytecode "checker", BV defines Sun's standard verifier, and LBC defines the requirements for constructing a certificate to satisfy LBV. We prove that the techniques in general provide equivalent type safety guarantees in general and in particular we show that lightweight verification is "tamper proof" in the sense that the type safety guarantee, provided by the checker at the execution platform, cannot be broken by crafting a "false" certificate or by inverting the bytecode.

We also explain how the verifier component of the KVM virtual machine, specifically targeted for small systems, and the on-card bytecode verifier by Leroy [26], targeted for Java Cards, can be simulated by the lightweight technique. (Indeed, the KVM verifier was directly derived from the initial presentation of this material: the pre-verifier corresponds to LBC and the runtime verifier to LBV with the certificate corresponding to the "stackMap" attribute.)

Finally, we present a prototype implementation of the lightweight verifier.

Keywords: Java Bytecode, type safety, static analysis, the Java virtual machine (JVM), mobile code, formal semantics, Internet, certificate, resource-constrained systems, KVM, Java Card.

Preface

Gentle reader, the present thesis is submitted as part of the requirements for obtaining *le diplôme de doctorat* (Ph.D.) of l'Université Denis Diderot (Paris 7), 2, Place Jussieu, 75251 Paris Cedex 05, France. The thesis work was actually carried out under the GIE Dyade project, at INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay, France. (Dyade is a joint venture between INRIA and Bull).

The *contributions* Section 1.1 specifies the technological and scientific contributions which this thesis has brought along. The remainder of Chapter 1 introduces the subject of the thesis and explain how the material is organized.

We assume that our reader is familiar with the Java language [19] and has some knowledge of the virtual machine [29], otherwise the thesis is intended to be self-contained for someone with a background in theoretical computer science and a certain amount of mathematical maturity. In order to make the thesis as accessible as possible for our target reader, we have devoted Chapter 2 to the presentation of the preliminary key subject areas which serve as the theoretical basis for the thesis. Consequently, the main chapters will include few definitions or explanations of external, theoretical origin. In order to help our reader to find specific notions and definitions, we have included lists of judgments, definitions, and figures, as well as an extensive index at the end of the document.

Acknowledgements. First of all I would like to thank the GIE DYADE project for having suggested the subject of this thesis. Also, I would like to thank INRIA Rocquencourt for hosting me and for providing the INRIA grant which made this thesis financially possible; in addition I would like to thank l'École Normale Supérieure de Lyon for hosting me during the first year as part of an exchange between DYADE and the PLUME group. Finally, my thanks go to l'Université Denis Diderot (Paris 7) for accepting me as a *doctorante* (Ph.D. student).

Then I would like to thank my supervisor, Professor Jean Goubault-Larrecq at ENS Cachan (l'École Normale Supérieure de Cachan) in Paris, for his talented, scientific criticism and his belief in me, even when our different scientific backgrounds seemed to create the most interesting (and developing) discussions during the thesis process. Also, I would like to thank the former head of the formal method team at DYADE, Dominique Bolignano, for having initiated this thesis, as well as Professor Pierre Lescanne who supervised me while I were at ENS-Lyon. But most of all I am proud to be able to thank the people who have kindly accepted to be part of my committee.

- Xavier Leroy, directeur de recherche, Projet Cristal, INRIA (Institut National de Recherche en Informatique et en Automatique), who has kindly accepted to be *rapporteur* of this thesis, has shown a particular interest in the outcome. His valuable comments in connection with

his own work on improving the bytecode verifier, has inspired me in my thesis work, and has lead to fruitful improvements in both contents and presentation of my ideas.

- Véronique Viguie Donzeau-Gouge, directeur de recherche, Centre de Recherche en Informatique de CNAM (Conservatoire National des Arts et Métiers), who has kindly accepted to be *rapporteur*, her strong presence in the formal proof community makes me grateful that she is willing to evaluate this work on practical formalization.
- Sophia Drossopolou, senior lecturer at the Department of Computing, Imperial College of Science, Technology and Medicine, who has kindly accepted to be *examineur* at the thesis defense, is an inspiring presence in the campaign for convincing the industrial Java community of why we should use formal methods and techniques to strengthen the foundation of “real-life” tools such as the Java platform in addition to having provided several important contributions in the area. I am grateful that she has accepted to be part of my committee.
- Thomas Jensen, Chargé de recherche, IRISA (Institut National de Recherche en Informatique et Systèmes Aléatoires), who has kindly accepted to be *examineur* at the thesis defense, has also contributed several of the major advances on formal methods for Java, especially in relation to the Java Card initiative that he is leading. I appreciate his insights and I am grateful that he is willing to participate in the Jury.
- Vincent Danos, Chargé de Recherches au CNRS (Centre National de la Recherche Scientifique), who has kindly accepted to be *examineur* at the thesis defense, is a major player in game theory and linear logic research. I look forward to his perspective especially on the resource consumption issues and I am grateful that he is willing to participate in the committee.

The thesis would not have been were it not for the continued encouragement of the community, notably the “Formal Underpinnings of Java” and “Formal Methods for Java Programs” workshops. Special thanks go to Thomas Jensen and “Action incitative Java Card” for inviting me to participate in the french Java Card workshops, and to Isabelle Attali for inviting me to Sophia-Antipolis to investigate the use of the TYPOL system to implement the static semantics.

I would like to thank both of my parents for providing me with the skills without which I could not have accomplished this task. My father, Viggo Madsen, for providing me with a profound compassion for systematics and mathematics, and my mother, Maja Corfitzen, for providing me with an urge to seek creativity and innovation. Also, I would like to thank my mother in law, Catherine Holm, for her help to make the french summary and presentation parts as linguistically correct as possible.

Finally, I would like to thank my husband Kristoffer Høgsbro Rose for a rewarding scientific collaboration on certain issues of this thesis, as accounted for in Section 1.1, not to forget his ever sharp comments, and \TeX nical assistance at moments where it was most needed; specifically, I thank him for the creation and help in using macro-packages to typeset diagrams and inference proofs, for setting up my \TeX and Java environment, and in general for practical and moral support during the last hectic months of thesis write-up.

Eva Rose
New York, August 2002

Contents

Preface	4
1 Introduction	9
1.1 Contributions	9
1.2 Motivation	13
1.3 Approach	16
1.4 Overview	21
2 Preliminaries	24
2.1 Basic Set Operations	24
2.2 Sorts and Structures	24
2.3 Orders	26
2.4 Inference Systems	27
3 The Formal Verification Context	29
3.1 The Machine Subset	29
3.2 The Context Components	39
3.3 The Class Hierarchy	46
3.4 The Example	47
4 Standard Verification Formalization	49
4.1 Analysis and Formalization Strategy	49
4.2 Bytecode Verification	63
4.3 Instruction Verification	67
4.4 Exception Verification	80
4.5 The Example	89
5 Lightweight Verification Formalization	93
5.1 Analysis and Formalization Strategy	93
5.2 Bytecode Verification	105
5.3 Instruction Verification	109
5.4 Exception Verification	115
5.5 The Example	120

6 Lightweight Certification Formalization	121
6.1 Analysis and Formalization Strategy	121
6.2 Bytecode Certification	126
6.3 Instruction Certification	132
6.4 Exception Certification	143
6.5 The Example	148
7 The Prototype Implementation	150
7.1 Implementation Strategy	150
7.2 The Class File Context	150
7.3 Lightweight Bytecode Verification	155
7.4 The Infrastructure Code	176
7.5 The User Manual	200
8 Comparisons	202
8.1 Sun’s “StackMap” Attribute	202
8.2 Leroy’s “On-Card Verifier”	203
9 Java Access Protection through Typing	206
9.1 Introduction	206
9.2 Read-only Field Access in Java	208
9.3 Field Access Types	209
9.4 Conclusion	211
10 Conclusions	212
10.1 Contributions	212
10.2 Related Work	212
10.3 Future Directions	214
A cksum () Example Details	215
A.1 Standard Verification Proof	215
A.2 Lightweight Verification Proof	221
A.3 Lightweight Certification Proof	226
A.4 Example Java Source and Bytecode	231
A.5 Execution of the lbv Program on Gcd11	234
List of Judgment Signatures	237
List of Definitions, etc.	238
List of Figures	245
Bibliography	247
Index	252

Chapter 1

Introduction

In Section 1.1, we summarize the contributions and development of the thesis. Then, in Section 1.2, we explain why we regard this study as being significant, followed by a presentation of our approach in Section 1.3 along with a canonical example which we shall refer to throughout the thesis. Finally, in Section 1.4, we give an overview of how the remaining document is structured.

1.1 Contributions

The main conceptual contribution of the thesis is the proposal of the *lightweight bytecode verification* technique for the Java programming language. The technique stages the “standard” Java bytecode verifier, *i.e.*, the officially specified verifier by Lindholm and Yellin [29, 30], into two phases in such a way that bytecode verification becomes feasible for code which is downloaded over a network onto a general, resource-constrained system.

Specifically, we give a formal presentation of lightweight verification for an important Java Virtual Machine (JVM) subset, and show how the technique provides the same type safety guarantees as standard bytecode verification on the same JVM subset. In particular we have, that these type safety guarantees are obtained without that the resource-constrained execution platform has to trust a third party, as it is the case when cryptography is applied.

In Figure 1.2 we show how the lightweight verification technique differs from a standard bytecode verification (as it performs over a network), which is depicted in Figure 1.1, in the way the two techniques are designed to perform over a network. The diagrams read as follows: arrows indicate the sequencing of events (in time horizontally and space vertically); firm rectangles show executing interpreters or compiler components, whereas rounded boxes denote code and other intermediate data entities.

Standard bytecode verification over a network, begins with the standard Java compiler,¹ which produces verified bytecode, but contains no record of the verification. Once the code is transmitted over an unsafe network to the code destination platform, it has to be (re)verified before execution. Lightweight bytecode verification also starts with a standard Java compiler, which produces verified bytecode. On the translation platform, however, lightweight verification begins with a *certification* (or pre-verification) step, which from already verified method bytecode produces a set

¹A standard Java compiler like Sun’s, includes bytecode verification as an integrated part.

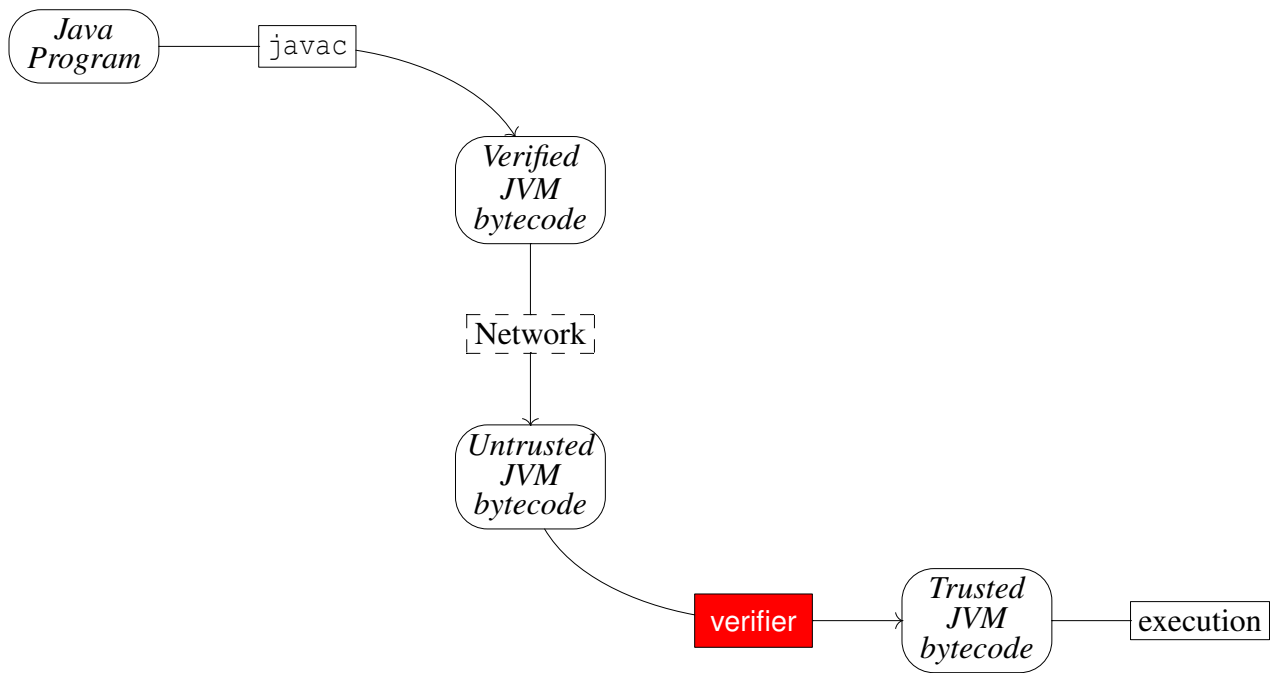


Figure 1.1: Standard bytecode verification.

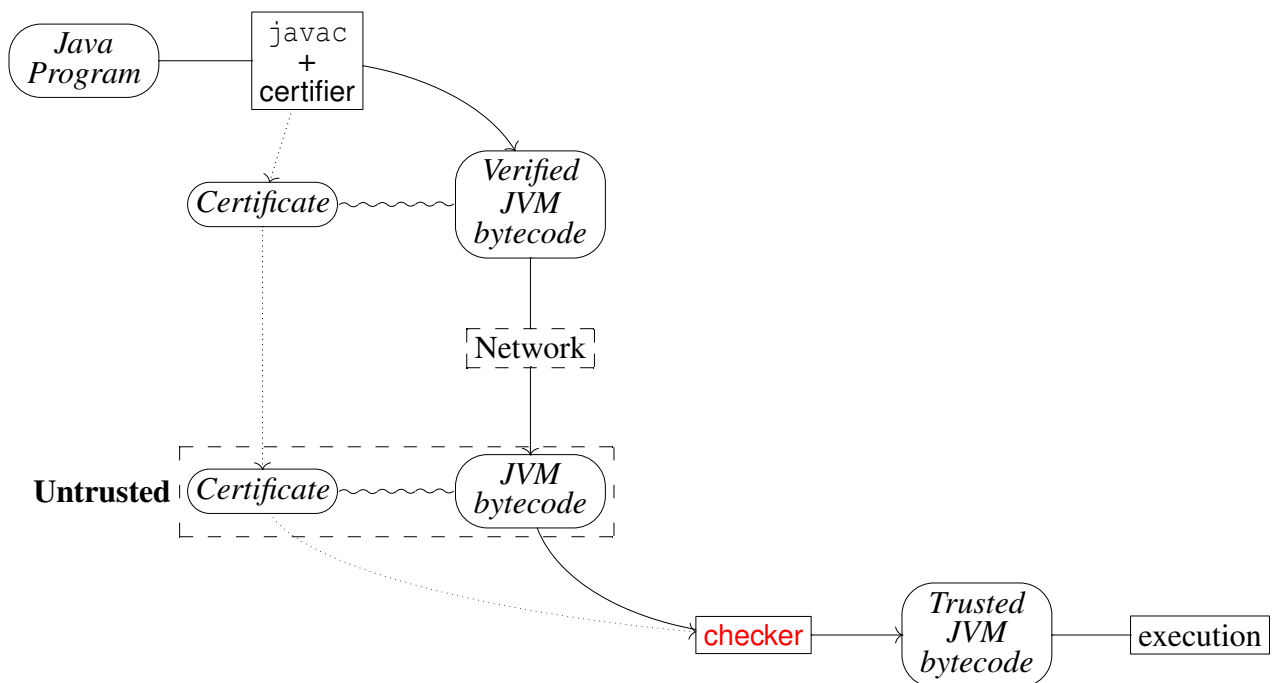


Figure 1.2: Lightweight bytecode verification.

of jump target type-annotations to accompany the method. At the execution platform, lightweight verification then proceeds with a *checker* step, which consists in checking that the accompanying type annotations actually are sufficient to guarantee, in one straight code pass, that the received method code is type safe at the execution platform.

The lightweight technique exploits the theoretical observation that Sun’s standard Java bytecode verifier implements a *fixed point iteration* over all possible execution paths (or traces) in some method code. (To be precise, the verifier is officially specified as a data-flow algorithm which in turn can be seen as a realization of a fixed point iteration strategy.) Something which implies that Sun’s standard bytecode verifier algorithm consumes scratch memory space which is proportional to the size of the code [29, §4.9.2], [58, 59].

The factor which makes lightweight verification feasible for resource-constrained platforms is the way that scratch memory space consumption is generally reduced in relation to standard bytecode verification: for a standard bytecode verifier, the verification of a method will build internal data structures described to be of size proportional to the code size [58, §5.3]. For the lightweight technique, the memory consumption during method verification is given in terms of a *certificate size* space and a *pending constraints* space. We shall show that these have size proportional to the number of reverse and forward jump targets (or “labels”), respectively, in the worst case. Specifically for Java Cards (and other embedded platforms) [9, 53, 57], the constraint space, consumed by standard bytecode verification, must all be updateable and thus stored in scratch memory. In contrast to this, the lightweight technique’s certificate is static and can be stored in flash memory² (along with the method bytecode) and only the pending constraints need to be stored in scratch memory.

The general idea, to stage bytecode verification with respect to the jump targets in the code, was outlined by the author [46, 47, Chapter 6], but never formalized up until this thesis work. A preliminary formalization was proposed in collaboration with K. H. Rose [48], which has later been mechanically verified to be correct by Klein and Nipkow [23]. The general lightweight verification idea [46, 47, Chapter 6] was subsequently implemented by Sheng Liang at Sun Microsystems as the foundation for KVM, the K-Virtual Machine’s “preverifier” and “stackmap” attribute [58, 59, personal communication]. The “preverifier” realizes the idea of the certification step, which establishes the certificate as *type annotations at jump targets*, given by the “stackmap” attribute. In this respect, Sun has modified the class file standard for the K-Virtual Machine to encompass a “stackmap” attribute, which today allows a class file to hold a method certificate. The lightweight verification formalization proposed in this thesis goes further than the above in formalizing essential parts of the JVM instruction set, and proves them to provide the same type safety properties as standard verification on that subset. Specifically, it goes further in reducing the size of the certificate as it was adopted by the KVM verifier (in a non-trivial way), and it goes further in showing how it generalizes the “on-card verifier” approach, recently proposed by Leroy [27], on our JVM subset.

In Chapter 8, we show that

- if the certificate contains type annotations for *every jump target* in the method code, then the lightweight verifier technique works as in the KVM verifier, but

²“Flash memory” is persistent memory, which can be read byte-wise, but only written to as a block of continuous data.

- if the code constraints of Leroy are imposed, *i.e.*, all local variables are initialized before any jumps, with no type changes throughout, and the stack being empty before statements, then the certificate is reduced to information about backward jump targets, which again is assumed to be available by Leroy’s algorithm. Thus, in this case there is no need for a certificate.

In order to provide the formal framework for presenting the lightweight verification idea and for showing the trust relation in the latter case, we present a formalization of standard bytecode verification in Chapter 4, based on the official verifier specification [29, 30]. To enhance the overview of the formal arguments, we have intended to keep down the complexity of such presentation by only formalizing a JVM subset. The subset has been selected and formalized in Chapter 3 to support an important subset of object-oriented features, notably those which has to do with jumps and abrupt code execution behaviour. The subset extends the one which has been considered in an earlier formalization of standard bytecode verification [47, Section 4.4] by the inclusion of several new aspects of the JVM, notably:

- One-dimensional array of integer and reference type.
- Exception raising and handling.

The actual lightweight verification formalization, which is original work to the thesis, is presented in two parts:

- lightweight certification, and
- lightweight checking, or simply, by abuse of notation, *lightweight verification*, as given in Chapter 6 and Chapter 5:.

The former shows how a lightweight bytecode verification certificate can be constructed from the type information gathered during a standard bytecode verification procedure, the latter shows how a method code is checked with a given certificate.

Based on the checker and verifier semantics, we state our main theorem, which formally shows that *the lightweight technique provides the same type safety guarantees as a standard bytecode verifier*. An important consequence is that the technique is *tamper-proof*: it is impossible to construct or modify a certificate to make the lightweight bytecode verifier accept bytecode that would not be accepted by the standard verifier.

In order to state a “proof of concept” for our formalized lightweight verification algorithm, we have implemented the lightweight bytecode verifier from Chapter 5, as a Java program which permits the lightweight verification of “real” Java `.class` files (as long as the implementation does not use JVM features outside our subset). The program uses Markus Dahm’s BCEL [12] to access `.class` files, but is otherwise an original contribution to the thesis. We present the documented Java code of the lightweight verifier with “the user’s manual” in Chapter 7.

Finally, at the end of the thesis, we have looked into the area of making dynamic type checks static in the context of small-resourced execution platforms. A study over permission types is presented in Chapter 9 in terms of an independent publication with K. H. Rose [49]. The paper proposes how the Java class file format could be slightly modified to include further static type information, allowing a bytecode verifier to check field access rights.

1.2 Motivation

Over the last years, there has been an increasing demand for generalizing the programming capability of small, independent, network-connected devices such as smart cards, point-of-sale terminals, PDAs, and set-top boxes, featuring an on-device microprocessor. Specifically with an interest in downloading foreign code onto those devices. The Java platform seems ideally suited for this task, as Java, and its predecessor Oak, were originally developed with just this type of deployment in mind [36, 41, 55]. The original prototype device included a verifier to provide safe program execution that did not need to trust the foreign code provider. For various reasons, the prototype failed in performance, but the idea survived in terms of a Java platform with an integrated abstract machine model (the JVM³) at which security issues could be addressed *independently* of the underlying execution platform, and *independently* of the code provider. Thus perfect for downloading arbitrary code from the Internet (that is a Java applet is loaded into a web browser.)

On the present Java execution platform, applet security is provided by the concept of “sand-boxing” [60]. Sand-boxing, among other things, features bytecode verification, which provides program type safety at the virtual machine level, without having to trust the code provider. Recently, the Java platform was reorganized to include a variant which specifically focuses on small-memory, resource-constrained environments featuring an on-device microprocessor [56, 57, 59]. The general problem of porting the existing sand-box technology onto sparsely resourced platforms, has specifically been the requirements for expensive memory. (For details on memory types we refer to Definition 1.3.7.) Bytecode verification, in particular, has traditionally been specified as a complex data-flow algorithm, which must store a set of static Java types for each (virtual) instruction of the method code being verified [29, 30, §4.9]. Even though some intermediate optimizations may be obtained in each method case, the complexity of the problem does dictate the general space performance.

Observation 1.2.1. A Java bytecode verifier, as officially described by Lindholm and Yellin, is specified to consume “scratch” memory which is proportional to the size of the method bytecode. By “scratch” we understand that each individual byte in the memory is addressable, *i.e.*, can be separately read from or written to. (For further details on memory types we refer to Definition 1.3.7.)

In the rest of this thesis, we shall denote any bytecode verifier with the described memory performance as “standard”.

Definition 1.2.2 (Standard Bytecode Verification). By a “standard bytecode verifier”, we mean any verifier algorithm, code, or verifier formalization, which adheres to the official verifier specification as described by Lindholm and Yellin [29, 30]

Remark 1.2.3. Even though the official 1999 specification [30] slightly extends the 1996 specification [29], they do not differ on the discussed instruction set.

Figure 1.1 on page 10, illustrates where standard bytecode verification is traditionally scheduled along a Java class file transfer over an untrusted network, in order to ensure Java safety at the code destination platform.

³“JVM” is an abbreviation for “Java Virtual Machine”.

When the present study began, it was in fact not obvious how to provide for bytecode safety on a resource-constrained code destination platform *without* depending on the approval from a third party, as is normally an announced strength of operating on an execution-independent level as the JVM. At that time, two conceptually different approaches were known, in some cases, to prevent safety and security violations at the code destination platform from foreign code, downloaded from an untrusted network: cryptography, or proof carrying code [37, 38]. Whereas a classic, cryptography-based approach was accessible for Java through digital signatures [60], the proof carrying code approach still remained unexplored with respect to Java. In order to evaluate these approaches, we begin by a comparison of the more significant virtues and drawbacks.

(Public key) cryptography. Techniques which implement this concept work by storing a “key” at the execution platform, distributed by a trusted party [5]. We list some pros and cons of the cryptographic approach:

- well-defined technology, already available for Java in terms of “digital signatures” [60],
- guarantees that the code is *literally identical* to the code which was sent off by the trusted party, however,
- the code consumer must trust an *external party* (the key provider),
- the number of keys to store has a tendency to grow, which is problematic on sparsely resourced devices, as *keys generally take up significant space* (for further examples, Anderson elaborates thoroughly on the problematic issue in a paper from 1994 [4]), and finally,
- the trust-relation between code consumer and code provider creates a *single point of failure* system. Whenever a device receives code which is widely distributed, it is an unfortunate situation when the device producer and the code provider do not necessarily trust the same sources (or each other). In banking, *e.g.*, this is the frequent case when credit cards are issued by one bank, whereas code is produced and encrypted by a competing bank.

Remark 1.2.4 (Digital signatures). Java today possesses the possibility to perform a checksum-based (public key) encryption of documents in terms of digital signatures of applet [31, p.146]. However, this does not avoid the other inconveniences which have been listed above.

Proof-carrying code (PCC). Implementations of this concept, which was first launched by Necula and Lee in 1996 [38], allows untrusted code to be statically verified as safe⁴ prior to execution, in the case where safe behavior can be logically defined and automatically verified. PCC works by the definition of a “security policy” expressed as a logical (type) system of decidable program safety properties. Thus, for a correct program, an additional formal prove (typing) can be constructed upon code transfer time, such that it can be mechanically decided whether the program adheres to the adopted policy or not at the code receiving platform. We list some pros and cons of the PCC approach:

⁴By the term safe program behavior, we understand behavior which does not allow access to private files/data, to overwrite important files/data, to access unauthorized resources, etc.

- The code consumer only has to trust its *own, internal security policy management*.
- The checker component implements a decidable proof check at the code receiving platform.
- The design and implementation of PCC methods (that is the formal definition of a security policy, proof generator, and mechanical proof checker) for different kinds of non-abstract machine-code frameworks have already yielded interesting optimization results [38].
- The number of transferred bytes will increase with a certificate.
- PCC methods only ensure that the code does not *circumvent* the code consumer's security policy. However, it does not ensure that the received code is identical to the code which was sent off.
- It is crucial that the proof producer and the code consumer have integrated the *same* safety policy in order for a correct program to be accepted. Thus, if the code emitter mistakenly sends an accompanying proof based upon a different (non-compatible) safety policy, the code will not get accepted, even in those cases where the program would have been safe to execute.
- In all existing descriptions of PCC, an operational semantics for the execution platform has been fully specified in a decidable, logical form such that safety properties can be specified, *i.e.*, defining a "Curry-Howard"⁵ style correspondence between the type system defined by the operational semantics and a generic logic. For object-oriented languages with sub-typing, such a correspondence is presently not known.

The first of the listed PCC arguments is a very strong one, if PCC should be sought applied for small, independent systems. By dealing with safety or security issues directly on these platforms, we simply obtain the ultimate protection against flaws in that no further trust relations are involved. This is in particular desirable for small Java platforms such as certain Java banking cards or military Java processing devices, where even a minimal risk of flaws may have fatal consequences. The second and third of the listed arguments are highly interesting, as a PCC checker component generally performs a simple, decidable type check of the transferred code, in order to proof that it meets given security measures at the execution platform. Clearly, if a PCC checker could be defined to ensure bytecode safety, it would be very likely to have a better space performance than a standard bytecode verifier, and thus make bytecode safety verification feasible for small devices. The fourth of the listed PCC arguments questions the applicability of PCC for space restricted platforms, as the number of bytes to be stored at the destination would increase if a PCC certificate became attached. We notice, however, that a PCC certificate doesn't have to be stored in scratch memory, as it only has to be addressable for reading. In comparison, the number of cryptographic keys which may need to be stored would have a tendency to grow with the amount of trust relations to maintain. (Even though such keys only have to be read, and as such can be stored on the same kind of "cheap" memory as PCC certificates, they generally do take up significant space.) The fifth argument states that whereas a cryptographic approach assures that the transmitted (bytecode safe)

⁵A "Curry-Howard" correspondence guarantees the existence of an isomorphism between a formal type system and a formal logic.

code remains literally identical, PCC makes a weaker assumption in that only the safety aspects of the received code would be evaluated. The last of the listed PCC arguments, however, questions the applicability of PCC, as it is presented by Necula and Lee [38], to ensure Java bytecode safety. As mentioned, the problem is that no one has yet found a way to specify an object logic which corresponds to the type systems of object oriented languages. We have,

Observation 1.2.5. The PCC methodology is not directly applicable for bytecode verification.

If an alternative deduction system to specify bytecode safety can be found, though, we may gain some of the discussed virtues of PCC, in particular that bytecode verification is assured directly on the code destination platform, without the involvement of a third party. This, however, will also require a reformulation of the certificate and checker concepts. Thus, instead of defining a certificate as a formal proof for some Java type safety policy, we simply look for a way to enrich the bytecode sufficiently, such that the complexity of the standard bytecode verification algorithm is reduced satisfactory. Finally, we will have to require that the simplified algorithm assures that bytecode type safety cannot be circumvented at the destination platform, if either unsafe code is received, or an inappropriate certificate enrichment is received. Hence, inspired by PCC, we will adopt the terminology of a “checker” for a simplified algorithm which makes use of enriched code, and a “certificate” for the actual code enrichment, provided that bytecode type safety cannot be circumvented at the code destination platform.

Generally we have, that for “small”, *i.e.*, sparsely resourced code destination platforms for transmitted code, a certificate will only need to be stored for *reading*, which requires relatively cheap memory (flash) as opposed to bitwise read/write memory types (scratch). A checker, however, will need regular read/write types of memory (scratch), just as any standard bytecode verifier algorithm, in order to store temporary data structures. (For a detailed explanation of the different memory types, we refer to Definition 1.3.7.) These considerations lead to the following important observation.

Observation 1.2.6 (Bytecode verification portability.). Assume that a bytecode-safety checker algorithm can be designed so that it has a simplified memory space complexity, without circumvention of type safety at the code destination platform. Then it is feasible that bytecode verification can be *ported* to a small platform.

1.3 Approach

When this study began, only Java 1.1 was yet released. In this thesis, we only consider Java bytecode type safety for Java 1.1 (thus not any of the additional features of Java 2 (1.2))

In order to provide the best proofs of concepts, we intend to primarily take a formal approach when re-designing the standard bytecode verifier for small platforms. As argued in relation to Observation 1.2.5, no general type-logic has been defined for object oriented languages. Instead, we propose to apply the general idea of PCC in defining a *formal proof system* in which bytecode type safety can be evaluated. Others have specified bytecode safety formally, when defining a static semantics for bytecode verification [7, 51]. Thus, by taking an operational approach to formalizing bytecode safety, we may not only profit from work done by others, but we obtain the deducted “type safety proofs” become stepwise specified. However, since we wish to reason over complete proofs,

we will adopt a “big step” approach as opposed to current bytecode verification specifications. By choosing Natural Semantics as our specification formalism, we obtain all of the above [14, 22, 34]. Consequently, bytecode verification becomes specified as logical type judgments where the type terms are given as members of algebraic sorts [63].

In order to begin a formalization of bytecode verification, we notice that this concept assures JVM type safety *per method*. Therefore we reformulate Milner’s original definition⁶ [33] of program type safety for JVM methods. Here, as a pseudo-formal, natural semantics judgment.

Definition 1.3.1 (JVM Type Safety). Let a JVM method be given by the bytecode sequence component C within a static JVM, compile-time type environment E . Furthermore, let \vec{x} denote a well-typed sequence of arguments to that method. Then the following implication must hold:

$$E \vdash_{\text{JVM}} C \text{ is well-typed} \Rightarrow \forall \vec{x} : C(\vec{x}) \text{ runs without type errors.}$$

which reads: when a method bytecode component C bytecode verifies within the static JVM compile-time type environment E , then C will run without type-errors on any well-typed argument sequence \vec{x} .

Remark 1.3.2. We notice that by a well-typed method argument sequence \vec{x} , we understand a sequence of arguments which matches the declared number, and the declared types. (The argument run-time types must be assignment compatible with the declared types [30, §2.5].) By the *static method type environment* E , we must refer to those statically typed JVM components which may affect the method’s well-typedness, *e.g.*, the class hierarchy, and reference to the current class.

The challenge is to formally express what it means for a method bytecode sequence to be statically “well-typed”. In Lindholm and Yellin’s description of Java bytecode type safety, the concept is specified by the existence of a set of (compile-time) Java types for all local variables and operand stack elements at each program (method) step, such that regardless of the control flow, no type constraints are violated [30, §4.9]. Theoretically, this corresponds to the existence of a fixed point⁷ on the sets of local variable and operand stack types which are associated to each method execution point, in the sense that the type-annotation set will satisfy all (compile-time) type constraints that the application of any potential execution path can enforce. In Lindholm and Yellin’s description of the Java (standard) bytecode verifier, it is specified by a data-flow algorithm, which in turn may be seen as a realization of some fixed point type-iteration strategy (over all method execution paths/traces.) [30, §4.9.1]. (The equivalent theoretical view of this is to consider standard bytecode verification as an abstract interpretation over the set of Java types [11].) It is very important to notice, however, that Java bytecode type safety does *not* depend on the existence of a minimal (or maximal), iterated fixed point (with respect to the JVM⁸ type-domain), but on the existence of *some* fixed point. The specification merely suggests that if the code is safe to execute, a fixed point will eventually be found by an adequate strategy. (An equivalent view is to say that if a set of type-annotations can be constructed by the abstract (compile-time) type analysis, then the code is well-typed/safe to execute.) The discussion leads to the following, important consideration.

⁶The type safety principle was originally formulated by Milner in terms of the famous phrase: “Well-typed programs cannot go wrong”.

⁷An element x of a function f ’s domain is a “fixed point” if $f(x) = x$.

⁸At the JVM level, a few Java types are not present, *e.g.*, booleans, which are compiled into integers.

Observation 1.3.3 (Type safety condition). We recall that method bytecode is regarded as type safe, if we can statically type-annotate the operand stack elements and the local variable table elements at each program point in the code, such that the instruction’s type constraints are not broken [30, §4.9]. Thus, the existence of such a set of type annotations is in itself sufficient to state that the method code is type safe.

Where we by “program point” refer to the relative byte position of each instruction in the method bytecode.

In connection with Observation 1.2.6, we discussed the possibility of defining an appropriate way to enrich network transmitted bytecode (by a “certificate”), such that type safety issues can be addressed in a space efficient way at the execution platform (by a “checker”). Three important properties have to be addressed in order to validate the approach.

Space efficiency A “checker” should run more space efficient on enriched bytecode than a standard bytecode verifier can run on “plain” bytecode, in order to be feasible on a space-limited platform.

Type safety guarantee We expect the same type safety guarantees from running a “checker” as from running a standard bytecode verifier.

Certificate size A “certificate” must have an appropriate size compared to the associated bytecode, or the improved space-performance may be contradicted.

In order to obtain a reduction of the standard verifier’s space complexity, we have to analyze what causes the excessive need for space. Hardly surprising, it is the control flow branches in the code, caused by, *e.g.*, a jump instruction, a jump subroutine instruction, an abrupt instruction, or (exception) throw instruction. which forces the data-flow analysis to perform another type-iteration over the code. In the general case, the standard verifier performs as a *type-reconstructor* which at each program point reconstructs a set of type annotations which obey the constraints which the code imposes for each new type-iteration. Since it is generally unpredictable where these instructions appear or where their eventual confluence target will be, the data-flow analyzer has to maintain a set of temporary type-annotations for each program point in the code until no subsequent control-flow branches add any changes to the type annotation set (theoretically, a fixed point is then reached over the set of type-annotations.).

An attractive guess for a certificate is to register only those type-annotations which are the result of diverting control-flow paths, and therefore cannot be straight forwardly evaluated. Based on the assumption that the strategy by which we verify the possible execution paths does not influence on the ability to decide on bytecode type safety, we suggest that type safety can be successively decided in each program point by *checking* the certificate type constraints, and by *checking* the type constraints imposed by the instruction at the previous program point. This strategy only requires one straight pass through the code, thus with a reduced space-complexity with respect to standard verification.

In our setting we initially imagine that method code can be standard verified at a “larger” code-provider platform. We imagine that this can be exploited when a standard verifier performs *successfully*, to process the resulting type solution set, a fixed point for the diverting type constraints in the code, to select an “appropriate” subset and transform them into a certificate. We believe that

such a certificate can be created in such a way that it provides the same safety guarantees when checked against the code at the destination platform, as standard bytecode verification.

Definition 1.3.4 (Java Lightweight Verification Concepts). We will specify a type safety checker as the “lightweight verifier”, or “checker” . The processed type-annotation subset is called the “certificate”, and finally, the selection and processing component for the “certifier”, or “certificate generator”.

In Figure 1.2 on page 10, we illustrated where these components are scheduled for a Java class transmission over an untrusted network.

Notation 1.3.5. We allow the term “lightweight verifier” instead of “lightweight checker”. Whenever the meaning is clear from the context, we will furthermore, by slight abuse of notation, use “lightweight verification” to denote the process of code certification, code enrichment, and code checking, for a successfully bytecode verified method.

The previous discussions lead to a specification of the semantical form for a type safety judgement, as used in the type safety definition.

Notation 1.3.6. Method bytecode type-safety will be specified as natural semantics judgments of the shape:

$$(1.3.6a) \quad \text{“environment”} \vdash \text{“method”}, \text{“typeannotations”}$$

Specifically, we shall label the turn-style with the relevant semantical signification, *i.e.*, \vdash_{bv} for standard bytecode verification, \vdash_{lbv} for lightweight bytecode verification (“checker”), and \vdash_{lbc} for lightweight bytecode certification, when needed to resolve any ambiguities.

We will consolidate the concept of lightweight verification, both formally and practically, in addressing the following matters:

Semantical correctness of standard bytecode verification semantics has to be established, as it serves as our formal basis for proving formal correctness for the lightweight semantics. The proof requires the specification of the dynamic semantics of JVM, together with an abstraction function and a realization function, formally showing that the semantics commute.⁹(For further details we refer to work by Cousot and Cousot [11], Despeyroux [14], and Nielson and Hankin [39].) As such proofs have already been thoroughly studied by others [45], and as the focus of the thesis is merely to present the improved bytecode verification technique, we will side-step such a formal correctness proof. We will, however, relate the static JVM semantic judgments on the fly to the official, operational description of the dynamic instruction behavior to provide for an intuitive understanding [30].

Tamper proof In order to ensure that lightweight verification does not introduce any security flaws, we must specifically prove that the lightweight bytecode verification technique is “tamper-proof”. By “tamper-proof” we mean that one cannot break the safety guarantee that the checker provides at the execution platform, by “clever” crafting of a certificate or inverting the bytecode (or both).

⁹Two semantics commute when it is possible to define an abstraction function α and a realization function γ between the semantics such that $\alpha \circ \gamma \leq \text{id}$ and $\gamma \circ \alpha \leq \text{id}$.

Proof of Concept We will investigate whether the lightweight formalization is practically feasible by implementation of a prototype of the checker component. A prototype which can run on “real” “.class” Java class files.

We will subsequently comment on the memory model, notably the different memory kinds which are feasible at the sparsely resourced, code destination platform.

Definition 1.3.7 (Memory Model). Since our general aim is to make lightweight verification feasible for smart cards, we assume a memory model which consists of two types: fast scratch memory, and slow flash memory, ROM, or EEPROM [9]. For our convenience, we shall briefly characterize each of these.

Scratch memory: General name for the part of the RAM which has the highest flexibility and permits read and write operations on individual bytes but the contents is lost when power is removed. Quite expensive, thus typically available in the range of a few thousand bytes on smart cards.

Flash memory: Persistent memory which can be read byte-wise, but only written to in large blocks of continuous data. Rather inexpensive, thus comes typically in a range of thirty to a hundred thousand bytes on smart cards.

ROM: Read-Only Memory that is created when the circuit is first manufactured. Early smart cards contained typically sixteen thousand bytes of ROM used for completely constant system programs and data, but this has become obsolete by the use of Flash memory even for system data (so it can be updated), except for the simplest cards produced in high volume.

EEPROM: “Electrically Erasable Read Only Memory” ROM that can be rewritten a limited number of times using a special but very slow process. Obsoleted by flash memory.

In the rest of the thesis, we will draw our conclusions based on flash memory, because it is the most common type of “slow memory” on smart cards, as well as the normal Java Card technology procedure for heap allocation on smart cards, which permit the objects to be stored as either persistent objects in the flash memory, or transient objects in scratch memory [9].

Observation 1.3.7a. Clearly, the *transferred method bytecode and lightweight certificate* should be placed in flash memory, as these components need to be written once (upon receipt) and read (bitwise), during lightweight verification. *Runtime data and other dynamic structures* which uses the object heap, however, may exploit the organization of a heap into persistent and transient objects, to reduce the need for scratch memory.

Finally, we present a Java source program which shall serve as our canonical example throughout the thesis. The program is made up for an imaginary situation where the compiled bytecode is downloaded onto a (Java) credit card from an external platform.

Example 1.3.8 (Checksum Computation). Assume that an algorithm is needed on a (Java) credit card to compute a checksum, based on the credit card number, and that it has to be externally downloaded. Specifically, we imagine the credit card number to be hard-coded on the credit card

together with methods to set and retrieve it, as part of some “credit card” package. The actual checksum definition is naturally hard-coded on the credit card as an abstract class, part of some “down-loadable” package. The downloaded checksum algorithm must be implemented as a non-abstract subclass (part of the “down-loadable” package), which implements the `cksum` method, and hence respects the abstracted type description. (A selection which already provides a certain level of security.) In Figure 1.3, we show an example of an implementation of such an algorithm in terms of the Euclidean “greatest common divisor”. Also, we expose the class which realizes the getter-method to fetch the hard-coded credit card number. (Notice, that we imagine the actual initialization of the credit card to be performed once and for all within a secured environment when the card is released.) The resulting `CrCardRd` object, which contains the credit card number, will thus reside on the persistent heap.

UnsetCrCard, Abort: define the exceptions to be raised if an error occurs.

```
class UnsetCrCard extends Exception {}
class Abort extends Exception {}
```

CrCardRd: contains and yields the credit card number.

```
class CrCardRd {
    int it; // credit card number or 0 if uninitialized
    public int getIt() throws UnsetCrCard {
        if (it == 0) throw new UnsetCrCard();
        return it;
    }
}
```

CkSum: subclasses of this abstract class will calculate an integer checksum based on the credit card number.

```
abstract class CkSum {
    abstract public int cksum(CrCardRd ccnum);
}
```

Gcd11: calculates the greatest common divisor of a credit card number, with 11 as divisor. The program is shown separately in Figure 1.3.

1.4 Overview

In this section, we present the overall structure the rest of the thesis. We notice, that a listing of the main contributions already has been given in Section 1.1, and that we in Example 1.3.8 presented the canonical example of this thesis.

We begin, in Chapter 2, with an explanation of notational conventions and symbols which are applied in definitions and specifications, together with a re-statement of some theoretical key-concepts upon which this work is based. In Chapter 3, we discuss how we have selected our

```

class Gcd11 implements CkSum {

    public int cksum(CrCardRd ccnum) throws Abort {
        int x;
        try {
            x = ccnum.getIt();
        } catch (UnsetCrCard e) {
            throw new Abort();
        }
        int y = 11;
        while (true) {
            int z = x - y;
            if (z > 0) { x = z; }
            else if (z == 0) { return x; }
            else { z = x; x = y; y = z; }
        }
    }
}

```

Figure 1.3: The `cksum()` source program.

representative JVM instruction subset, we account for which features are not encompassed. We discuss and formalize the notion of a static verification context, and we informally summarize the officially specified instruction behaviour along with their syntactical formalizations. In Chapter 4, we present the semantics of standard bytecode verification. We discuss our formalization approach to type safety for the JVM, which leads to the definition of a JVM type safety lattice on which our verification semantics are based. Then we formalize standard bytecode verification as a type safety checker with respect to a supposed type solution set. Our formalization of static instruction verification, is based on the official specification, which we use as our reference throughout the presentations. In Chapter 5, we present the formal semantics of lightweight bytecode verification. We discuss the lightweight idea and a formal strategy for its realization. The technique is finally specified as a type safety checker semantics with respect to a type certificate. In Chapter 6, we present the formal semantics of lightweight certification. In particular we discuss and define what to understand by a lightweight type certificate. The certifier is then specified as a certificate generating type safety checker with respect to a supposed type solution set. In Theorem 6.1.1, we finally formulate and prove that lightweight bytecode verification provides the *same* safety guarantees as standard bytecode verification. Finally, we draw the conclusion that lightweight verification is tamper proof in that no “false” certificate can be invented which makes type unsafe code pass the lightweight verifier. In Chapter 7, we present a prototype implementation of the lightweight verifier. The implementation can run on “real” “.class” Java class files with handcoded certificates. The implementation is completed with a user manual. Finally, some example programs are presented. In Chapter 8, we compare lightweight verification with the “on-card” algorithm by Xavier Leroy, and with Sun’s KVM bytecode verifier. For each of these algorithms, we study how the lightweight

verifier technique coincide, when we impose the same assumptions as in those cases. In Chapter 9, we present an article on how to statically perform access-modifier check, which today are done at runtime, by a small addition of compile-time information. In Chapter 10, we finally conclude over the obtained results. In particular, we relate the work to other, similar studies, and we discuss the importance and the future perspectives of the thesis. Finally we notice Appendix A, which largely contains examples of hand-unfolded verification proofs for a canonical JVM programming example, as well as example-runs with the lightweight verification implementation.

Chapter 2

Preliminaries

This chapter enumerates the way we use standard formal notations and definitions in the thesis. We present several standard results from the literature without proof.

2.1 Basic Set Operations

The following basic operations are used. (Some examples are used and explained in the following sections.)

Definition 2.1.1 (Boolean Logic). We use usual boolean logic operators: if a and b are boolean expressions then $a \wedge b$ is the logical and of the two, $a \vee b$ is the inclusive logical or of the two. $a \Rightarrow b$ is the logical implication of b from a , $a \Leftrightarrow b$ is true if a and b are logically equivalent (sometimes written “iff”), and $\neg a$ is true if a is false. Furthermore, $\forall x, p(x) : q(x)$ is true if *all* x satisfying the predicate $p(x)$ also satisfy the predicate $q(x)$, $\exists x, p(x) : q(x)$ is true if *some* x satisfying the predicate $p(x)$ also satisfy the predicate $q(x)$, and $\exists! x, p(x) : q(x)$ is true if *exactly one* x satisfying the predicate $p(x)$ also satisfies the predicate $q(x)$; in all three cases x may denote more than a single variable and we abbreviate “ $x, p(x)$ ” to just “ $p(x)$ ” if the identity of x is obvious.

Definition 2.1.2 (Sets). We use basic set notation: $\{e_1, e_2, \dots\}$ denotes the set containing the members e_1, e_2, \dots , $e \in S$ is true if e is a member of the set S , $\{f(e) \mid p(e)\}$ denotes the set containing the members generated by the expression $f(e)$ for each e that satisfies the predicate test $p(e)$, $S_1 \cup S_2$ denotes $\{e \mid e \in S_1 \vee e \in S_2\}$, $S_1 \cap S_2$ denotes $\{e \mid e \in S_1 \wedge e \in S_2\}$, $S_1 \setminus S_2$ denotes $\{e \mid e \in S_1 \wedge e \notin S_2\}$, and \emptyset is the empty set $\{\}$. The subset test $S_1 \subseteq S_2$ is true when $\forall e \in S_1 : e \in S_2$, sets are equal $S_1 = S_2$ when $S_1 \subseteq S_2 \wedge S_1 \supseteq S_2$, and $S_1 \subset S_2$ if $S_1 \subseteq S_2 \wedge \neg(S_1 \supseteq S_2)$. Finally, $\mathbf{P}(S)$ is the “power set” of subsets $\{S' \mid S' \subseteq S\}$.

Definition 2.1.3 (Integers). We use the usual set of natural numbers $n \in \mathbf{N} = \{0, 1, 2, \dots\}$ and integers $\mathbf{Z} = \{\dots, -1, 0, 1, 2, \dots\}$ with the usual arithmetic and comparison operations.

2.2 Sorts and Structures

In order to specify the signature of operators and composite structures we declare sorts using the following algebraic conventions [32, 63, 50].

Definition 2.2.1 (Sorts Declarations). A component of a structure is defined by declarations such as

$$(2.2.1a) \quad v \in \text{Sort Definition}$$

which declares v as the “sort metavariable” for the sort Sort with a carrier set defined by the *Definition*.

The *Definition* is either “=” followed by a simple set, or “::=” followed by a BNF-style structural definition (where “::=” reads “has the structure” and $|$ denotes a choice between structures, each written as an example value using metavariables of the appropriate sorts for the inductively included sorts).

Structural definitions imply the operator signature of the involved operators.

Example 2.2.2. The following two definitional equations occur in the thesis:

$$(3.2.2b) \quad \text{FREF} \in \text{FieldRef} ::= \text{fieldref}(\text{CID}, \text{ID}, \text{T})$$

$$(3.2.6a) \quad \text{M} \in \text{Method} = \text{MethSig} \times \text{ReturnType} \times \text{ExcAtt} \times \text{CodeAtt}$$

These specify the following points:

- FREF , FREF' , $\text{FREF}_{\text{hello}}^3$, *etc.*, are metavariables denoting objects of sort FieldRef .
- Elements of the FieldRef sort can be constructed by the fieldref operator with the signature $\text{ClassIdent} \times \text{Ident} \times \text{Type} \rightarrow \text{FieldRef}$ (derived from the fact that the metavariables CID , ID , and T have those respective types as can be verified through the index.)
- M is a metavariable denoting an object of sort Method wherever it occurs.
- The Method sort contains quadruples of the form $\langle \text{MSIG}, \text{RT}, \text{EA}, \text{CA} \rangle$ as the metavariables MSIG , RT , EA , and CA , denote members of the sorts MethSig , ReturnType , ExcAtt , and CodeAtt , respectively.

Definition 2.2.3 (Tuples and Sequences). We use $\langle e_1, \dots, e_n \rangle$ to denote the n -tuple of the elements e_1 through e_n . For sets S_1, \dots, S_n , $S_1 \times \dots \times S_n$ and $\prod_{i=1}^n S_i$ both denote the set of tuples $\{ \langle e_1, \dots, e_n \rangle \mid e_1 \in S_1 \wedge \dots \wedge e_n \in S_n \}$. S^n is a shorthand for $\prod_{i=0}^n S$, and S^* denotes $\bigcup_{n=0}^{\infty} (\prod_{i=0}^n S)$, and the special symbol ϵ denotes the *empty sequence* $\langle \rangle \in S^0$ for any S . *Tuple concatenation* is denoted $s_1 \cdot s_2$; by abuse of notation we will write $e \cdot s$ and $s \cdot e$ for s , as well as e instead of $\langle e \rangle$, when this is unambiguous

Definition 2.2.4 (Functions). $f \in A \rightarrow B$ denotes that f is a total function mapping each member x of the “domain” set $\text{Dom}(f) = A$ to an element of the “range” or “codomain” $\text{Co}(f) = B$ denoted $f(x)$. Functions can also be written as the set of their simple mappings $\{x_1 \mapsto f(x_1), \dots, x_n \mapsto f(x_n)\}$ or $\{x_i \mapsto f(x_i) \mid i \in \{1, \dots, n\}\}$. In case the function domain is implied then we also allow functions to be defined by a case list like

$$f(x) = \begin{cases} y_1 & \text{if } x = x_1 \\ y_2 & \text{otherwise} \end{cases}$$

Finally, $f \in A \xrightarrow{\text{part}} B$ denotes that f is a partial function mapping just *some* of the elements of A into elements of B .

2.3 Orders

This section explains our use of partial orders [63, 50].

Definition 2.3.1 (Partial Ordering). A relation (\sqsubseteq) is a subset of $S \times S$ with the special notational convention that we write $x \sqsubseteq y$ for $\langle x, y \rangle \in (\sqsubseteq)$. Furthermore, \sqsubseteq is a partial order if it is *reflexive*: $\forall x \in S : x \sqsubseteq x$, *antisymmetric*: $\forall x, y \in S : (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$, and *transitive*: $\forall x, y, z \in S : (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$.

Definition 2.3.2 (Partially Ordered Sets). We will assume that every set S is equipped with a partial ordering \sqsubseteq_S . Unless an ordering is explicitly specified for the set a set is assumed equipped with member equality corresponding to the “flat” partial ordering.

Some generic derived sets are equipped with special partial orders (it is easily seen that they are, indeed, partial orders):

1. Given S is equipped with the partial order \sqsubseteq_S . S_\perp is the set S *lifted* to contain the special new element \perp_S and be equipped with the partial order defined by $x \sqsubseteq y$ iff $x = \perp_S \vee x \sqsubseteq_S y$.
2. Given S is equipped with the partial order \sqsubseteq_S . S^\top is the set S *sunk* to contain the special new element \top_S and be equipped with the partial order defined by $x \sqsubseteq y$ iff $x \sqsubseteq_S y \vee y = \top_S$.
3. Given sets S_1 and S_2 equipped with the partial orders \sqsubseteq_1 and \sqsubseteq_2 , respectively. Then $S_1 \times S_2$ is equipped with the pointwise partial ordering \sqsubseteq defined by $\langle x_1, x_2 \rangle \sqsubseteq \langle y_1, y_2 \rangle$ if $x_1 \sqsubseteq_{S_1} y_1 \wedge x_2 \sqsubseteq_{S_2} y_2$.
4. Given sets A and B where B is equipped with the partial order \sqsubseteq_B . Then $A \rightarrow B$ is equipped with the function ordering \sqsubseteq defined for $f_1, f_2 \in A \rightarrow B$ by $f_1 \sqsubseteq f_2$ iff $\forall a \in A : f_1(a) \sqsubseteq_B f_2(a)$.

When clear from the context we omit the subscript in each case.

Definition 2.3.3 (Join and Meet). The following operators are defined on sets with a partial order \sqsubseteq :

1. $x \sqcup y$ is the *join* (or *greatest lower bound*) of the elements $x, y \in S$ defined, if it exists, as the $z \in S$ such that $x \sqsubseteq z, y \sqsubseteq z$, and $\forall v \in S : (x \sqsubseteq v \wedge y \sqsubseteq v) \Rightarrow z \sqsubseteq v$.
2. $x \sqcap y$ is the *meet* (or *least upper bound*) of the elements $x, y \in S$ defined, if it exists, as the $z \in S$ such that $z \sqsubseteq x, z \sqsubseteq y$, and $\forall v \in S : (v \sqsubseteq x \wedge v \sqsubseteq y) \Rightarrow v \sqsubseteq z$.

We shall use similar rotated symbols for other orderings, *e.g.*, the intersection \cap is the “meet” of the usual \subseteq subset relation.

Definition 2.3.4 (Lattice). A set S equipped with the partial order \sqsubseteq is a *lattice* if, for all $x, y \in S$, both $x \sqcup y$ and $x \sqcap y$ exist and are members of S . If only \sqcup (\sqcap) exists then it is an upper (lower) semi-lattice; in each case it is *complete* if every subset $S' \subseteq S$ has a unique $\bigsqcup_{x \in S'} x$ and/or $\bigsqcap_{x \in S'} x$ in S (as appropriate).

Proposition 2.3.5 (Preservation of Lattice Property). When the constituent partial orders have a lattice property then all the derived sets constructed in Definitions 2.3.2 have the (same) lattice property.

Proposition 2.3.6 (Completion of semicomplete lattice). If S with \sqsubseteq is a complete lower semi-lattice with a greatest element then it is a complete lattice.

Example 2.3.7 (Partially Ordered Sets). \mathbf{N} is equipped with the flat partial ordering $=$. $\mathbf{N}_{\perp}^{\top}$ is a lattice since all members are less than \top and larger than \perp so $n_1 \sqcap n_2$ and $n_1 \sqcup n_2$ always exist, respectively; similarly $\mathbf{N}_{\perp}^{\top} \times \mathbf{N}_{\perp}^{\top}$ is also a lattice since there all members are less than $\langle \top, \top \rangle$ and larger than $\langle \perp, \perp \rangle$.

Notation 2.3.8 (Function Meet and Join). The meet (\sqcup) and join (\sqcap) of functions is only defined for functions of the same sort $A \rightarrow B$. For $f \in A \rightarrow B$, $a \in A$, and $b \in B$ we define the following special function abbreviations:

$$(2.3.8a) \quad (f \sqcap \{a \mapsto b\})(a') = \begin{cases} f(a') \sqcap b & \text{if } a' = a \\ f(a') & \text{otherwise} \end{cases}$$

$$(2.3.8b) \quad (f \sqcup \{a \mapsto b\})(a') = \begin{cases} f(a') \sqcup b & \text{if } a' = a \\ f(a') & \text{otherwise} \end{cases}$$

2.4 Inference Systems

We shall use structural operational semantics [44] with inference rules in the “big step” style used for static semantics in natural semantics [14, 22] and for the static semantics of Standard ML [34].

Definition 2.4.1 (Judgments). A *judgment* is a formula defined by inference rules. The judgment signature defining the signature of well-formed judgments is given in a box. An *inference rule*

$$\frac{\text{premise}_1 \cdots \text{premise}_k}{\text{conclusion}} \text{ (Rule) where side conditions}$$

Specifies that a proof of each of the judgments premise_1 through premise_k constitute a proof of the conclusion judgment provided the side conditions all hold. The tree obtained by matching premise judgments with the (sub)proof of each is called a *proof tree*. Finally we remark that meta-variables used in inference rule definitions are assumed locally bound for that rule but in actual proofs all metavariables must be bound in the proof context.

Example 2.4.2. Definition 4.2.7 contains the judgment signature box

$$\boxed{\text{StdContext} \vdash_{bv} \text{Method}, \text{FrameTypeApprox}}$$

which formalizes that we must give inference rules for building proofs for a subset of the possible combinations of judgment terms of the form $\Gamma \vdash_{bv} M, \text{FTA}$ with Γ , M , and FTA denoting members of the sorts StdContext , Method , and FrameTypeApprox , respectively.

The proof of Proposition 4.5.1 contains the fragment

$$(2.4.2a) \quad \frac{\frac{\overline{CH^{ck} \vdash FTA_0^{ck} \sqsubseteq FT_0^{ck}} \quad (4.1.28a) \quad \frac{\boxed{A.1.2}}{\Omega^{ck} \vdash 0, C^{ck}, \emptyset \Rightarrow PPS^{ck}} \quad (4.2.8b)}}{\Gamma^{ck} \vdash M^{ck}, FTA^{ck}} \quad (4.2.7a)}$$

where all the ck-indexed metavariables as well as the subproof $\boxed{A.1.2}$ are defined in the proof context.

Chapter 3

The Formal Verification Context

In Section 3.1, we discuss how to select a representative JVM instruction subset and how it is formally presented. (We also include a brief, informal summary of these instruction’s runtime semantics.) In Section 3.2 we discuss and specify a formal type context for bytecode verification together with a formalization of a class file. In Section 3.3 we formalize the notion of a Java subclass hierarchy, and in Section 3.4, we consider our canonical example.

3.1 The Machine Subset

We select a representative instruction subset for the Java 1.1, except for the jump routines as argued below. This means that the additional features of Java 2 (1.2) such as inner classes and reflection, will not be considered. The subset is chosen to represent all important type safety concerns, notably non-trivial, object oriented language features. It is large enough to write meaningful programs, though small enough for the reader to maintain an overview throughout the formalizations. Generally speaking, we have weighted the inclusion of instructions which support *object features*, e.g., heap object creation and manipulation, instance method invocation, instance field access, and *control flow statements*, e.g., jump statements, abrupt statements, and exception handling. We have left out support for several language features, notably:

Jump subroutines given by the `jsr` instruction, has been shown to introduce many complications in semantics and typing [52] yet are only intended to support the Java “`try...finally`” construct so their use is in practice strikingly small; indeed a study by Freund [16] found full unfolding of `jsr` to extend the average class file insignificantly, for example a mere 0.02% for the JDK. Perhaps for that reason it is already common practice in some commercial Java compilers to unfold these as part of normal compilation. Therefore, they will not be encompassed by our formalizations.

Class methods, initializers and class fields are not supported. Even though the compile-time type verification of initializers and class fields do not differ from the verification of instance methods¹ and instance fields, they do require additional syntax to be included in our formalizations. They are, however, a trivial matter to add.

¹Class methods and class initializers are syntactically simpler in that they do not contain the self-reference (`this`).

Access modifiers and packages are not supported by our JVM subset since access modifiers are not part of the type system but checked at link (or “resolution”) time; in Chapter 9, however, we discuss how these checks could be added to the type system and thus the bytecode verifier.

Interfaces are not supported by our formalization, but are left for future work.

Dead code is not encompassed by our formalization. A relatively harmless restriction, as dead code does not interfere with the execution of a program, and thus with bytecode verification.

The standard Java environment includes the `java.lang` package as a default. In our formalizations we have omitted packages. Instead we simply assume that the `Object`, `Throwable`, `Runtime` and `Exception` classes are included in the bytecode verification environment with their standard subclass relationship.

Before we proceed, we briefly comment on the type system at the JVM level.

Observation 3.1.1 (The JVM type system). Generally speaking, the Java type system at the source level is the type system for which we evaluate bytecode type safety. There are, however, a few differences which influence our type formalizations:

- First, the Java types “boolean” and “char” are not part of the JVM level type system, as they are translated into integers. (*Arrays* of boolean and char are, however, allowed by the JVM.)
- Second, there is an insufficient mapping between the specification of multidimensional arrays at the Java source level and that of the JVM level: whereas Java multidimensional array types may have an infinite number of dimensions, JVM limits the number of dimensions to a finite number because class file format leaves just one (unsigned) byte to declare the dimension of a multidimensional array.

In the rest of this thesis, we shall write “the JVM type system”, or “the Java type system at the JVM level”, whenever we need to emphasize the difference from the Java language type system.

Definition 3.1.2 (A representative data-type subset). We have selected a set of JVM types which we consider the most important for bytecode verification.

Primitive types: integers of type `int`.

Non-primitive types: object references: class types or one-dimensional arrays of references and `int`. The classes `Object`, `Throwable`, `Exception`, and `RuntimeException` are built-in.

We notice that class (reference) types are sufficient to cover the type description of abstract classes, as they only exist by their name (that is a class reference into the constant pool.)

For reasons of clarity, we have not included all JVM data-types.

Definition 3.1.3 (Omitted data-types). We have left out those data-types which do not raise additional verification-questions other than already addressed by the representative data-type set.

Primitive types: Other numeric and character types: `boolean`, `char`, `float`, `double`, `byte`, `short`, and `long`. These can all be added fairly easily as bytecode verification only checks these types as the values are loaded into or stored from local variables where the only problem is managing double-word values.

Non-primitive types: Multidimensional arrays (of more than one dimension). As mentioned in Observation 3.1.1, however, these compound types constitute a finite type set at the JVM level, which each can be formalized as one-dimensional array types and may thus be added easily to our formalizations. We also do not deal with the arrays of primitive type other than `int` (`boolean`, `char`, `float`, `double`, `byte`, `short`, and `long`).

Interfaces: constitute a special set of data-types at the JVM level. Interfaces do not directly get validated as they do not contain bytecode; as mentioned before we will not consider interfaces further.

We finally present a JVM instruction subset of 32 instructions, which we consider as sufficient to serve as our representative JVM target machine for the Java language subset which we have discussed so far in this section. The instruction subset is close to the target machine for Nipkow and von Oheimb's Java subset BALI [40] (hardly surprising, as both language subsets were selected for allowing non-trivial, object-oriented Java programs). Each virtual machine instruction is officially specified by an opcode byte, followed by zero or more operand byte [30, §6]. We formalise the opcode byte by the instruction's official, mnemonic description, whereas the operand bytes (if any), are formalized by their numeric value.

Definition 3.1.4 (The Formal Target Machine). The representative JVM instruction subset is formally specified.

$$(3.1.4a) \quad I \in \text{Ins} = \text{OpCode} \times Z \cup \text{OpCode}$$

$$(3.1.4b) \quad \text{OP} \in \text{OpCode} = \{\text{dup}, \text{pop}, \text{iadd}, \text{isub}, \text{iconst}_0, \text{iconst}_1, \text{aconst_null}, \text{istore}, \text{astore}, \text{iload}, \text{aload}, \text{iastore}, \text{aastore}, \text{iaload}, \text{aaload}, \text{newarray_int}, \text{anewarray}, \text{arraylength}, \text{checkcast}, \text{ldc_w}, \text{new}, \text{getfield}, \text{putfield}, \text{invokevirtual}, \text{ifne}, \text{ifle}, \text{ifnull}, \text{goto}, \text{athrow}, \text{return}, \text{ireturn}, \text{areturn}\}$$

Notation 3.1.5 (Instruction representation). In the rest of this thesis, we will write '`OP`' for an instruction without any argument bytes, and '`OP[n]`' for an instruction which has the number value `n` for its argument bytes. Notice how we have made the implicit byte operands explicit in our formalization. In particular, we adopt that a jump instruction's argument bytes construct a signed code jump offset. By convention we state that if $n \leq 0$ we have a backward jump, whereas $n > 0$ indicates a forward jump.

Along with a systematic presentation of each formalized instruction, we informally present their run-time semantics in Figure 3.1 through Figure 3.6, based on the official specification by Lindholm and Yellin [30, §6.4] with the following parameter conventions.

Notation 3.1.6 (Informal Description Parameters). The following parameter names are used in all of the informal instruction descriptions.

value: Value of type as described in text.

int: Value of integer type.

ref: Reference to class instance (object).

arrayref: Reference to array instance (object).

length: Integer used as length of array.

index: Index into constant pool or array.

arg: Method argument.

branch: Branch offset.

Furthermore we define that when *index* is used to indicate a two-byte instruction operand, then $index_1$ and $index_2$ will respectively denote the first (high) and second (low) (8-bit) byte of the (16-bit) index bytes. Similarly for *branch*.

Since all of Figure 3.1 through Figure 3.6 have been equally structured, they can be read in the same way.

Notation 3.1.7 (How to read the tables). In the first column we list an instruction’s opcode and its operand bytes (if any). In parentheses, following each instruction, its size is given in bytes. In the second column, we show the instruction’s execution effect on the current operand stack. (Notice, that the operand stacks grow towards the right as in the official specification [30, §6.4].) Finally, in the third column, we generally describe the instruction effect in plain English.

We group instructions by their common characteristics and properties.

Definition 3.1.8 (Stack Instructions). The “stack” instructions operate exclusively on the stack, where they manipulate the stack top values. Thus, this instruction group do not expect an operand byte. In Figure 3.1, these stack manipulations are described in detail.

$$(3.1.8a) \quad \text{StackIns} = \{ \text{iconst}_0, \text{iconst}_1, \text{aconst_null}, \text{dup}, \text{pop}, \text{iadd}, \text{isub} \}$$

Definition 3.1.9 (Local Variable Instructions). The “local variable” instructions operate on both the local variable table and the stack, manipulating the values stored in the local variable table, as these cannot be directly accessed. The instructions expect a one-byte operand to index the variable table. Their runtime behavior is described in Figure 3.2.

$$(3.1.9a) \quad \text{LocalIns} = \{ \text{istore}, \text{astore}, \text{iload}, \text{aload} \}$$

We state an important property for this instruction group.

instruction (size)	stack before \Rightarrow after	Description
iconst_0 (1)	... \Rightarrow ..., 0	Pushes a 'zero' integer.
iconst_1 (1)	... \Rightarrow ..., 1	Pushes a 'one' integer.
aconst_null (1)	... \Rightarrow ..., null	Pushes a 'null' reference.
dup (1)	..., <i>value</i> \Rightarrow ..., <i>value</i> , <i>value</i>	Duplicates the stack top (an integer or reference value).
pop (1)	..., <i>value</i> \Rightarrow ...	Removes the stack top (an integer or reference value).
iadd (1)	..., <i>int</i> ₂ , <i>int</i> ₁ \Rightarrow ..., (<i>int</i> ₁ + <i>int</i> ₂)	Adds the two stack-top integers.
isub (1)	..., <i>int</i> ₂ , <i>int</i> ₁ \Rightarrow ..., (<i>int</i> ₁ - <i>int</i> ₂)	Subtracts the second stack-top integer from the true stack top integer.

Figure 3.1: The stack instruction's runtime behavior.

instruction (size)	stack before \Rightarrow after	Description
istore <i>index</i> (2)	..., <i>int</i> \Rightarrow ...	Pops the top integer into local variable number <i>index</i> .
astore <i>index</i> (2)	<i>ref</i> , ... \Rightarrow ...	Pops the top reference into local variable number <i>index</i> .
iload <i>index</i> (2)	... \Rightarrow ..., <i>int</i>	Pushes the integer from local variable number <i>index</i> .
aload <i>index</i> (2)	... \Rightarrow ..., <i>ref</i>	Pushes the reference from local variable number <i>index</i> .

Figure 3.2: The local variable instruction's runtime behavior.

instruction (size)	stack before \Rightarrow after	Description
iaload (1)	$\dots, arrayref, index$ $\Rightarrow \dots, int$	Pushes an integer from the indexed location of the referenced integer array.
aaload (1)	$\dots, arrayref, index$ $\Rightarrow \dots, ref$	Pushes a reference value from the indexed location of the referenced (class) reference array.
iastore (1)	$\dots, arrayref, index, int$ $\Rightarrow \dots$	Stores the stack-top integer at the indexed location of the referenced integer array.
aastore (1)	$\dots, arrayref, index, value$ $\Rightarrow \dots$	Stores a well-typed stack-top reference value at the indexed location of the referenced (class) reference typed array.
newarray_int (2)	$\dots, length$ $\Rightarrow \dots, arrayref$	Pushes a reference to a new integer array heap allocation, of the stack-top given length.
anewarray <i>index</i> (3)	$\dots, length$ $\Rightarrow \dots, arrayref$	Pushes a reference to a new (class) reference array heap allocation of a stack-top given length, where the (class) reference is described at the constant pool location $index_1 \times 256 + index_2$.
arraylength (1)	$\dots, arrayref$ $\Rightarrow \dots, length$	Pushes the length of the stack-top referenced array.

Figure 3.3: The array instruction’s runtime behaviour.

Observation 3.1.10 (Local variable invariance). Local variable values are only accessed or modified through the stack.

Definition 3.1.11 (Array Instructions). The “array” instructions mainly creates and manipulate array structures through the stack, thus require no operands. For `anewarray`, however, type information is also fetched from the constant pool; this instruction therefore expects a two-byte index operand. We refer to Figure 3.3 for a detailed description of their runtime behavior.

$$(3.1.11a) \quad \text{ArrayAccessIns} = \{\text{iaload}, \text{aaload}, \text{iastore}, \text{iaload}, \\ \text{newarray_int}, \text{anewarray}, \text{arraylength}\}$$

Remark 3.1.12 (Formalization of `newarray`). This JVM instruction expects a one-byte operand to indicate the (primitive) type of its elements [30, p.343]. Since we only operate with one primitive type `int` in our formal setting, we formalize this directly into its mnemonic description as `newarray_int`. However, we maintain the original instruction length of 2 bytes (which includes the type-indicator operand.)

Definition 3.1.13 (Constant Pool Instructions). The “constant pool” instructions operate both on the constant pool and on the stack, where they basically fetch or compare with the constant items which are kept in the constant pool. Consequently, they expect a two-byte index operand. For a detailed runtime description of these instructions, we refer to Figure 3.4.

$$(3.1.13a) \quad \text{ConstPoolIns} = \{\text{checkcast}, \text{ldc_w}, \text{new}, \text{putfield},$$

getfield, invokevirtual }

Remark 3.1.14 (Formalization of new). At runtime, execution of the new instruction is not sufficient to allocate a class instance on the object heap, until an initializing constructor is called: the new instruction must be combined with an invokespecial call of the special constructor method `<init>`. In order to simplify our formalizations, however, we have implicitly abstracted heap allocations and initialization issues away in this setting, as they have no effect on bytecode verification. For more details on the subject we refer to Freund and Mitchell [17].

In order to formalize those instructions which may cause a jump in the code, we shall characterize them according to whether they definitely cause a specific jump (the goto instruction), whether they cause one of two jumps (the branch instructions), and, finally, all instructions which may throw an exception to be caught by some exception handler. In Section 4.4 we describe the exception-caused code jump situations. For the present, we simply formalize the instruction syntax for goto and branch instructions.

Definition 3.1.15 (The Branch Instructions). The “branch” instructions operate both on the code and the stack, where they will perform one of two possible jumps in accordance with the “flag” value on the operand stack. The jump offset is given by a two-byte index operand. We refer to Figure 3.4 for their runtime descriptions.

(3.1.15a) $\text{BranchIns} = \{\text{ifne}, \text{ifl}, \text{ifnull}\}$

Definition 3.1.16 (The Goto Instruction). The “goto” instruction operates exclusively on the code, where it performs an unconditional jump in accordance with the offset given by a two-byte index operand. The runtime behavior is described in Figure 3.5.

(3.1.16a) $\text{GotoIns} = \{\text{goto}\}$

Abrupt instructions may cause the current method execution to terminate. We split their definitions in two: instructions which *may* terminate the current execution (*i.e.*, `athrow`) and those which *always* terminate the current execution (*i.e.*, return instructions).

Definition 3.1.17 (The athrow Instruction). The “athrow” instruction operates on the stack and on the method frame for the exception which is thrown. Thus, it takes no operands. In Figure 3.6 we describe the runtime behavior.

(3.1.17a) $\text{ThrowIns} = \{\text{athrow}\}$

Definition 3.1.18 (Return Instructions). The “return” instructions terminate the current method execution, regardless of the machine state, thus take no operands. They partly operate on the stack. As the current frame is deleted, however, the operand stack becomes discarded. The runtime behavior is described in Figure 3.6.

(3.1.18a) $\text{ReturnIns} = \{\text{return}, \text{ireturn}, \text{areturn}\}$

instruction (size)	stack before \Rightarrow after	Description
checkcast <i>index</i> (3)	\dots, ref $\Rightarrow \dots, ref$	Verifies that the stack-top object reference is null or can be cast to something of the reference type, described at the constant pool location $index_1 \times 256 + index_2$.
ldc_w <i>index</i> (3)	\dots $\Rightarrow \dots, int$	Pushes the integer at the constant pool location $index_1 \times 256 + index_2$.
new <i>index</i> (3)	\dots $\Rightarrow \dots, ref$	Pushes the class reference to the new object, given from the constant pool location $index_1 \times 256 + index_2$.
getfield <i>index</i> (3)	\dots, ref $\Rightarrow \dots, value$	Pushes an instance field value in the stack-top referenced class instance, where the field is given by the field descriptor at the constant pool location $index_1 \times 256 + index_2$.
putfield <i>index</i> (3)	$\dots, ref, value$ $\Rightarrow \dots$	Stores the value at the stack-top in the object, referenced by the second stack-top element. The value type must be well-typed with respect to the field descriptor at the constant pool location $index_1 \times 256 + index_2$.
invokevirtual <i>index</i> (3)	$\dots, ref, [\dots arg_1]$ $\Rightarrow \dots, [value]$	Invokes the method described at the constant pool location $index_1 \times 256 + index_2$. The stack-top arguments arg_1, \dots (if any), must be well-typed with respect to the method's formal parameter descriptions. The class instance reference, given by the subsequent stack argument, must indicate a subclass to the invoked method's descriptor class. The instruction may push a return value, which must be well-typed, if any, with respect to the method descriptor's return type.

Figure 3.4: The constant pool instruction's runtime behavior.

instruction (size)	stack before \Rightarrow after	Description
<code>goto <i>branch</i></code> (3)	\dots $\Rightarrow \dots$	Evaluation continues at another instruction in the method code, relative to the <code>goto</code> opcode given by a (signed) offset ($branch_1 \times 256 + branch_2$) without affecting the current frame.
<code>ifne <i>branch</i></code> (3)	$\dots, value$ $\Rightarrow \dots$	For a non-zero numeric value at the stack-top, evaluation will continue at another instruction in the method code, relative to the <code>ifne</code> opcode position, given by the (signed) offset ($branch_1 \times 256 + branch_2$). In any case, the value is popped.
<code>ifle <i>branch</i></code> (3)	$\dots, value$ $\Rightarrow \dots$	For a numeric value at the stack-top which is less or equal to zero, evaluation continues at another instruction in the method code, relative to the <code>ifle</code> opcode position, given by the (signed) offset ($branch_1 \times 256 + branch_2$). In any case, the value is popped.
<code>ifnull <i>branch</i></code> (3)	\dots, ref $\Rightarrow \dots$	For a non-null class reference at the stack-top, evaluation continues at another instruction in the method code, relative to the <code>ifnull</code> opcode position, given by the (signed) offset ($branch_1 \times 256 + branch_2$). In any case, the value is popped.

Figure 3.5: The jump instruction's runtime behavior.

instruction (size)	stack before \Rightarrow after	Description
return (1)	\dots \Rightarrow discarded	Indicates return from a method with return type <code>void</code> . Any values in the current frame are discarded.
ireturn (1)	$\dots, value$ \Rightarrow discarded	Indicates return from a method with an integer return type <code>value</code> at the stack-top. The primitive return type must be identical to the method's declared return type. Any values in the current frame are discarded.
areturn (1)	\dots, ref \Rightarrow discarded	Indicates return from a method with an object reference return type <code>ref</code> at the stack-top. The return type must be well-typed with respect to the method's declared return type. Any values in the current frame are subsequently discarded.
athrow (1)	\dots, ref $\Rightarrow ref$	Indicates that an exception of class instance type <code>ref</code> has been thrown. (A reference for <code>Throwable</code> or one of its subclasses.) If a matching exception handler is found in the current method, program evaluation continues at the location of the handling code on a cleared operand stack where only the exception reference has been pushed. Otherwise, the entire method frame is popped.

Figure 3.6: The abrupt instruction's runtime behavior.

Remark 3.1.19 (The Java Card and the J2ME instruction sets). The described instruction set constitutes a subset² of the Java Card language instruction set, and the J2ME, *i.e.*, Java 2 Micro Edition, instruction set [56, 57].

Example 3.1.20 (The checksum bytecode representation.). The original source code for the `cksum()` method is shown in Figure 1.3. The disassembled bytecode, which has been translated by `javac` JDK version 1.1 [54], is shown in Figure 3.7, and the `cksum()` bytecode is formalized by the code component C. (The full Java source program and its disassembled output are shown in an appendix in Section A.4.)

```
(3.1.20a)      aload[1] invokevirtual[1] istore[2] goto[+8] pop
                new[2] athrow ldc_w[3] istore[3] iload[2] iload[3]
                isub istore[4] iload[4] ifle[+10] iload[4] istore[2]
                goto[-16] iload[4] ifne[+6] iload[2] ireturn
                iload[2] istore[4] iload[3] istore[2] iload[4]
                istore[3] goto[-39]
```

First we notice that approximately half of the instructions are given in terms of their quick-versions. Since quick-instructions are not generally available in the considered subset, we have unfolded these instructions into their standard opcode and operand part by hand. This with the side-effect that the program point offsets and thus the values in the exception table become slightly bigger. Second, the compiler-generated jump and branch targets are given as absolute addresses; each of these has been replaced by their relative counterparts in the formalized code. Third, the instruction sequence 9-13 in Figure 3.7, which was generated by the translation of the Java source statement “`new()`;” is given the formalized translation ‘`new`’ under the restrictions in Remark 3.1.14. Finally, `bipush` has been replaced by the more general `ldc_w` instruction.

3.2 The Context Components

Even though bytecode verification is performed per method, the actual method code is transmitted in units of *class files*, which contains the methods to be verified [30, §4.1]. The information carried by a class file thus becomes part of its method’s verification context. We formalize a class file as the method’s verification context, given by the sort “`ClassFile`”. A (sub)class hierarchy is the result of a subtle interaction between class resolution, class loading, and bytecode verification [30, §5.3–5.4]. It is the *class resolver* which actually constructs the class hierarchy by allocating the necessary space on the object heap during method evaluation, whereas the *class loader* takes care of fetching the class files. Bytecode verification performs in between, since the Java runtime system requires that a method is verified before it can be run. Notice, however, that even though class resolution and class loading happens in between bytecode verification, they are not part of the verifier. (For a fuller treatment of class loading formalization, we refer to work by Jensen et al. [21].) Even though the class hierarchy may not have been loaded at once during verification, we conclude that all necessary class information on the (compile-time) hierarchy structure is available

²Java Card integers are not guaranteed to be 32 bit, thus Java Cards are strictly speaking not JVM platforms. Since this doesn’t affect bytecode verification, we tacitly side-step the observation.

```

public int cksum(CrCardRd);
    /* Stack=2, Locals=5, Args_size=2 */
    0 aload_1
    1 invokevirtual #9 <Method int getIt()>
    4 istore_2
    5 goto 17
    8 pop
    9 new #1 <Class Abort>
    12 dup
    13 invokespecial #7 <Method Abort()>
    16 athrow
    17 bipush 11
    19 istore_3
    20 iload_2
    21 iload_3
    22 isub
    23 istore 4
    25 iload 4
    27 ifle 36
    30 iload 4
    32 istore_2
    33 goto 20
    36 iload 4
    38 ifne 43
    41 iload_2
    42 ireturn
    43 iload_2
    44 istore 4
    46 iload_3
    47 istore_2
    48 iload 4
    50 istore_3
    51 goto 20
Exception table:
    from   to   target type
     0     5     8   <Class UnsetCrCard>

```

Figure 3.7: The `cksum()` method bytecode.

at verification time. As an approximation, we formally assume that a completion of the referenced class hierarchy is fully available during verification. We formalize this component by the sort name “ClassHier”. In (3.2.1a) we have summarized these decisions with our formal definition of a “standard verification context”. In this section we proceed with a top-down specification of the ClassFile sort, whereas the specification of ClassHier is postponed to Section 3.3.

First we formalize those ClassFile items³ which are relevant to bytecode verification (within our subset restrictions). These are the constant_pool[], the this_class, and the super_class items [30, §4.4]. The fields[] item is only relevant for resolution, whereas the method[] item actually contains the method being verified, thus is not considered part of its own context). Since the superclass can be looked-up in the class hierarchy, however, and we assume that the class hierarchy is fully available during verification, the formal information provided by the super_class item becomes obsolete in the method’s verification context. In (3.2.1b) we have thus formalized a class file by its constant pool table (constant_pool[]) and its class identity item (this_class). Finally, we notice that a constant pool is specified as a table, which straight forwardly formalizes as a (partial) map from table entry numbers to constant pool items, as depicted in (3.2.1c).

Definition 3.2.1 (The Formal Verification Context).

$$(3.2.1a) \quad \Gamma \in \text{StdContext} = \text{ClassFile} \times \text{ClassHier}$$

$$(3.2.1b) \quad \text{ClassFile} = \text{ConstPool} \times \text{ClassIdent}$$

$$(3.2.1c) \quad \text{CP} \in \text{ConstPool} = \mathbf{N} \xrightarrow{\text{part}} \text{Item}$$

$$(3.2.1d) \quad \text{IT} \in \text{Item} ::= \text{CID} \mid \text{FREF} \mid \text{MREF} \mid \text{int}$$

In the rest of the thesis we shall explicitly refer to elements stored in the constant pool as *constant pool items*.

A constant pool typically serves as a common repository for large constants or constant information which is shared between several methods [30, §4.4]. With the restrictions we have imposed on our subset, we consider the constant pool item CONSTANT_Class_info which holds a class reference, the constant pool item CONSTANT_Fieldref_info which holds a reference to a (name-and-type) field description, the item CONSTANT_Methodref_info which holds a reference to a (name-and-type) method description, or the item CONSTANT_Integer_info which holds a reference to an integer constant. The CONSTANT_Class_info item is for example used by the new instruction to create an instance of the indicated class. It is formalized by the unspecified name sort “ClassIdent”. (We deliberately ignore issues arising from class loaders that require the management of class names with a class loader part and a “local” name part. For the verifier’s perspective, these are simply considered as different classes with different names [21]). The CONSTANT_Fieldref_info item is for example used by the putfield and the getfield instructions to access the indicated field. It is formalized by the FieldRef sort. The CONSTANT_Methodref_info item is for example used by the invokevirtual instruction to invoke the indicated method. It is formalized by the MethRef sort. Finally, the CONSTANT_Integer_info item is straight forwardly formalized by the int sort. In (3.2.1d) we have summarized these decisions by the definition of a constant pool item as an element in a four-sorted algebra. We will continue to specify each of these sorts in a top-down manner.

³An “item” is a field in a JVM class file structure [30, §4.1].

Definition 3.2.2 (Constant Pool Items).

- (3.2.2a) $CID \in \text{ClassIdent}$
(3.2.2b) $FREF \in \text{FieldRef} ::= \text{fieldref}(CID, ID, T)$
(3.2.2c) $MREF \in \text{MethRef} ::= \text{methref}(CID, MSIG, RT)$
(3.2.2d) $MSIG \in \text{MethSig} ::= \text{methsig}(ID, T^*)$
(3.2.2e) $RT \in \text{ReturnType} ::= \text{void} \mid T$
(3.2.2f) $T \in \text{Type}_{CH} = \text{Type}_{CH, \text{prim}} \cup \text{Type}_{CH, \text{ob}}$
(3.2.2g) $T_{CH, \text{ob}} \in \text{Type}_{CH, \text{ob}} = \text{Type}_{CH, \text{cref}} \cup \text{Type}_{CH, \text{aref}}$
(3.2.2h) $T_{CH, \text{prim}} \in \text{Type}_{CH, \text{prim}} ::= \text{int}$
(3.2.2i) $T_{CH, \text{cref}} \in \text{Type}_{CH, \text{cref}} = \text{ClassIdent}|_{CH}$
(3.2.2j) $T_{CH, \text{aref}} \in \text{Type}_{CH, \text{aref}} ::= T_{CH, \text{prim}} \square \mid T_{CH, \text{cref}} \square$
(3.2.2k) $ID \in \text{Identifier}$

Remark 3.2.3. Because a class name unambiguously⁴ identifies a specific class reference into the constant pool, we have formalized a class as a member of the unspecified name sort `ClassIdent`.

Notation 3.2.4 (The class hierarchy subscript). By ‘CH’ we indicate a specific class hierarchy as specified in Notation 3.3.3. When ‘CH’ is used as a subscript to a sort, we indicate a restricted sort with respect to the class hierarchy indicated by ‘CH’. The `ClassIdent|CH` sort, *e.g.*, specifies the finite set of class names which (unambiguously) identify the classes in ‘CH’ (without structure.)

Elements in `FieldRef` or `MethSig` are specified as tagged triples in (3.2.2b) and (3.2.2c), which unambiguously identify a field or a method, respectively. These triples specify the *class* where the field or method is declared, their unique “*member*⁵ *identification*”, and their declared *type*. For fields, the member-identification is the field *name*; for methods, it is the method’s declared *signature* which uniquely defines the method in some class. (Method signatures are special as they dynamically decide which method to dispatch, whereas fields are statically dispatched.) Method signatures formalize as tagged tuples of the method’s declared *name*, and (finite) list of declared *parameter types*, as specified in (3.2.2d). Field and method names are easily formalized by the unspecified name-sort `Identifier` in (3.2.2k). Types are formalized within the type restrictions described in Definition 3.1.1. In (3.2.2e) to (3.2.2j), types are generally formalized as elements in `Type`, within the current class hierarchy which is formally given by `CH`. These are primitive types, class instance types, or (one-dimensional) array types. The only primitive type in our type model is the integer type, which formalizes straight forwardly as the unary type element `int`, regardless of the available class hierarchy. Class instance types are identified by their class names, which formalizes straight forwardly by the `ClassIdent` sort within the current class hierarchy `CH`. Similar considerations apply for array types. Finally, the `void` method return type is specifically being formalized as the unary type element `void`.⁶

⁴A “class name” here means a distinguishable name with respect to a fully qualified name.

⁵A member of some class is a field or method, declared in that class.

⁶JVM uses a compact string representation for `void`.

Example 3.2.5 (The checksum class file verification). We formalize the `Gcd11` class file as a verification context component for the `cksum()` method. The method was described in Example 1.3.8.

$$\begin{aligned} \text{CP} &= \{1 \mapsto \text{methodref}(\text{CrCardRd}, \text{methsig}(\text{getIt}, \epsilon), \text{int}), \\ &\quad 2 \mapsto \text{Abort}, \\ &\quad 3 \mapsto \text{int}\} \\ \text{CID} &= \text{Gcd11} \end{aligned}$$

From the (disassembled) `cksum()` output in Figure 3.7, we notice that the #-references in the code actually refer into the current constant pool. In our formalization of the constant pool `CP`, we let entry 1 map to the description of `getIt()` (entry #9), and entry 2 to the description of `Abort` (combining entry #1 for the class and #7 for the initialization). (Notice how we use ϵ to formalize that `getIt()` has no formal parameters.) Finally, we map entry 3 to the specification of an integer constant. The self-reference tag, `CID`, is formalized by the class name `Gcd11` of the referenced instance type to be.

We proceed with the formalization of a method within our formalization restrictions. Declared methods reside inside the class file as `method_info` structures. Each of these is specified by a `name_index`, a `descriptor_index`, and an `attribute_info` field [30, §4.7]. These fields identify and completely characterize the method. Within our formalization restrictions, the two first fields, *i.e.*, the method identifier and return type, is given by `MethSig` and `ReturnType`. The last field specifies the method's attribute structures. The method code is specified in the `Code_attribute` and the exceptions which might be throw, in `Exceptions_attribute`. We formalize these attributes as the `CodeAtt` and `ExcAtt` sorts. (The sort names formalizes in fact the attribute identification tag fields.) Since these are the only attributes which the JVM specification requires to be recognized by a JVM implementation, we do not consider other method attributes in our formalization. In Definition 3.2.6, we have listed our formalization decisions. The exception attribute is specified by `exception_index_table[]` [30, §4.7.4]. As the table numbers are insignificant to bytecode verification, the table simply formalizes as a list of exceptions, given by their (class) names.

The code attribute is specified by a multi-tuple of fields, *i.e.*, a `attributes[]` field, `max_stack` and `max_locals` fields, a `code_length`, a `code[]` fields, and finally an `exception_table[]` field [30, §4.7.3]. Since the `attributes[]` only contains runtime information, however, we have ignored the field in our formalizations. The `max_stack` and `max_locals` fields of the code attribute, indicate the maximal capacity of the operand stack and local variable table of the current frame to be. If we side-step the formalization of the program and frame counters, the formalization of the capacity fields can be regarded as a formalization of the frame capacity given by the `MaxFrame` sort. The `code_length` and `code[]` fields indicate the code length and instruction list. Since the former field is not used during bytecode verification, and since the length appears from the number and length of the instructions stored in the `code[]` table, it is indirectly formalized as the side condition in the program point semantics of Definition 3.2.10. Thus, only the code field is explicitly formalized by the `Code` sort, as specified in Definition 3.2.8. Finally, the `exception_table[]` field is formalized in Definition 3.2.12 as the list `ExcHandlers`.

Definition 3.2.6 (A Method).

$$(3.2.6a) \quad M \in \text{Method}_{\text{MS,ML}} = \text{MethSig} \times \text{ReturnType} \times \text{ExcAtt} \times \text{CodeAtt}_{\text{MS,ML}}$$

- (3.2.6b) $EA \in \text{ExcAtt} = \text{ClassIdent}^*$
(3.2.6c) $CA \in \text{CodeAtt}_{MS,ML} = \text{MaxFrame}_{MS,ML} \times \text{Code} \times \text{ExcTable}$
(3.2.6d) $MFR \in \text{MaxFrame}_{MS,ML} = \text{MaxStack}_{MS} \times \text{MaxLocals}_{ML}$
(3.2.6e) $MS \in \text{MaxStack}_{MS} = 0..65535$
(3.2.6f) $ML \in \text{MaxLocals}_{ML} = 1..255$

Remark 3.2.7. The `max_stack` field holds the maximal number of (word-sized) elements on the operand stack, whereas the `max_locals` holds the maximal number of expected local variable assignments, each number given by two unsigned bytes. Notice that since we only consider instance methods, this is set in the local variable table, $ML \geq 1$ for our subset. As we cannot extend the table index byte without the `wide` instruction, `MaxLocals` is only supported up to 255.⁷ Because max frame constraints are assured by an initial file format check prior to bytecode verification [30, §4.8.1], these numbers have been integrated directly.

The `code[]` field of the `Code_attribute` structure specifies “bytecode”, and is implemented as an array of instruction bytes. It is formalized by the `Code` sort in Definition 3.2.8 as a (non-empty) list of formalized instructions. In particular we formalize the (possibly empty) subsequences of this list by the `CodeSeq` sort.

Definition 3.2.8 (Bytecode Formalization).

- (3.2.8a) $C \in \text{Code} = I \cdot CS$
(3.2.8b) $CS \in \text{CodeSeq} = I^*$
(3.2.8c) $PPS \in \text{PPoints} = \mathbf{P}(\text{PPoint})$
(3.2.8d) $PP \in \text{PPoint} = 0..65535$

Remark 3.2.9. The maximal code index is stored in the `code_length` field in four (unsigned) bytes. However, Java imposes a static constraint on the `code_length`, which limits the number of addressable code bytes to 65535 [30, §4.8]. This limit is part of the initial file format check (prior to bytecode verification) [30, §4.8.1]. However, it is a type safety requirement that *execution cannot fall off the end of the code* [30, §4.9.2]. As pointed out by Leroy, this code limit is in practice not generally enforced by the JDK 1.2 verifier (for example when a method does not contain exception handlers, as exception handler locations are addressed directly by a two-byte program point field), we read the official bytecode specification by Lindholm and Yellin [30, §4.9.2] as though this is merely an optimization in the JDK 1.2. We have therefore integrated this upper-limit into our code formalization. Since a method code array is specified as non-empty [30, §4.8.1], we have equally integrated this lower-limit in the formalization of the `Code` sort.

Definition 3.2.10 (Bytecode Sequence Program Points). Let C be the bytecode of a method, and let CS be a consecutive bytecode sequence which contains the last instruction of C . The set PPS_C is defined as the set that satisfies the judgment $\vdash 0 : CS, \emptyset \rightarrow PPS_C$ given by the rules:

- (3.2.10a)
$$\frac{}{\vdash PP : \epsilon, PPS \rightarrow PPS}$$

⁷Java Cards in fact support only 31 local variable allocations.

$$(3.2.10b) \quad \frac{\vdash PP' : CS, PPS' \rightarrow PPS''}{\vdash PP : I \cdot CS, PPS \rightarrow PPS''} \quad PP' = PP + \text{len}(I)$$

where $\text{len}(I)$ is the length in bytes of the instruction I .

Notation 3.2.11. In the rest of this thesis, we shall write PPS_C when we explicitly want to denote the set of program points which occur in the bytecode sequence C .

The `exception_index_table[]` field of the `Code_attribute` field is formalized by the sort `ExcTable` in Definition 3.2.12. Each of these elements, formalized by the `ExcHandler` sort, implements an exception handl, *i.e.*, a translation of a `try-catch` clause at the Java source level. Since only the order of the exception handlers in `exception_index_table[]` is significant to the static verification of the handle search procedure [30, §3.10], the table is simply formalized as a list. In JVM, an exception handler field is given as a quadruple item of the `start_pc` and `end_pc` fields, the `handler_pc` field, and the `catch_type` field. The `start_pc` and `end_pc` fields mark-off the instruction range where exceptions may be thrown during method execution, thus formalizes as `TryRange`. The `handler_pc` indicate the program point of the handling code, which formalizes as the program-point sort `CatchHandle`. The `catch_type` field, limits the kind of exceptions which can be handled. Within the restrictions imposed on our subset, a `catch_type` is a class reference type which is previously formalized by the `ClassIdent` sort.

Definition 3.2.12 (The Exception Handler Table).

$$(3.2.12a) \quad ET \in \text{ExcTable} = \text{ExcHandlers}$$

$$(3.2.12b) \quad EHS \in \text{ExcHandlers} = \text{ExcHandler}^*$$

$$(3.2.12c) \quad EH \in \text{ExcHandler} = \text{TryRange} \times \text{CatchHandle} \times \text{CatchType}$$

$$(3.2.12d) \quad TR \in \text{TryRange} = \text{PPoint} \times \text{PPoint}$$

$$(3.2.12e) \quad CH \in \text{CatchHandle} = \text{PPoint}$$

$$(3.2.12f) \quad CT \in \text{CatchType} = \text{ClassIdent}$$

Remark 3.2.13. Constraints on the code attribute format are ensured by an initial well-formedness format check, which is not encompassed by the bytecode verifier [30, §4.8.1] The start and end of a “try-range”, *e.g.*, are required to be valid indices to an opcode in the associated method code, and is met by our formalization by assuming that the range start and end values are to be found among the associated method’s program points.

Example 3.2.14 (The checksum method formalization). We formalize the `method_info` structure for the `cksum()` method from Example 1.3.8.

$$\begin{aligned} M &= \langle \text{MSIG}, \text{RT}, \text{EA}, \text{CA} \rangle \\ \text{MSIG} &= \text{methsig}(\text{cksum}, \langle \text{CrCardRd} \rangle) \\ \text{RT} &= \text{int} \\ \text{EA} &= \langle \text{Abort} \rangle \\ \text{CA} &= \langle \text{MFR}, \text{C}, \text{ET} \rangle \end{aligned}$$

where the code attribute parameter, CA , is listed below.

$$\begin{aligned} \text{MFR} &= \langle 2, 5 \rangle \\ C &= \text{aload}[1] \text{ invokevirtual}[1] \text{ istore}[2] \text{ goto}[+8] \text{ pop} \\ &\quad \text{new}[2] \text{ athrow ldc.w}[3] \text{ istore}[3] \text{ iload}[2] \text{ iload}[3] \\ &\quad \text{isub istore}[4] \text{ iload}[4] \text{ ifle}[+10] \text{ iload}[4] \text{ istore}[2] \\ &\quad \text{goto}[-16] \text{ iload}[4] \text{ ifne}[+6] \text{ iload}[2] \text{ ireturn} \\ &\quad \text{iload}[2] \text{ istore}[4] \text{ iload}[3] \text{ istore}[2] \text{ iload}[4] \\ &\quad \text{istore}[3] \text{ goto}[-39] \\ \text{ET} &= \langle \langle \langle 0, 7 \rangle, 10, \text{UnsetCrCard} \rangle \rangle \end{aligned}$$

The bytecode parameter, C , was already formalized in Example 3.8. We refer to this example for a detailed explanation of the code formalizations.

3.3 The Class Hierarchy

Generally speaking, one can relate any loaded class to its superclass through the class reference given by the `super_class` item. A reference which is available in any class file (except for the `Object` class). Whence, it can be justified to talk about a “class hierarchy”. Since each class only can (`super_class`) reference one parent class and Java does not permit class hierarchy cycles, we have that *the class hierarchy constitutes a tree* with `Object` as its root and “proper” subclasses as its non-root nodes. In order to formalize the notion of such a class hierarchy, we adopt the binary subclass order-relation “ \leq ” as defined by Abadi and Cardelli [1]. The relation is defined as the transitive and reflexive closure of a parent map (except for `Object`, tagged by the identity map).

Definition 3.3.1 (The Class Hierarchy). A class hierarchy is formalized as follows.

$$(3.3.1a) \quad \text{CH} \in \text{ClassHier} = \mathbf{P}(\text{ClassIdent})$$

where the ordering $\langle \text{CH}, \leq \rangle$ is defined for the *finite sets* in $\mathbf{P}(\text{ClassIdent})$, only. We furthermore assume that `Object` (the root) is part of any class hierarchy $\langle \text{CH}, \leq \rangle$.

Remark 3.3.2. The definition of an ordering only for finite sets of class names, corresponds to the reality in that the number of classes which are loaded or linked into the class hierarchy during program execution in practice is finite.

Notation 3.3.3. By slight abuse of notation, we let CH indicate an subclass ordered, finite set $\langle \text{CH}, \leq \rangle$ in the rest of this thesis.

For a class based object oriented language as Java, there is a close correspondance between the Java subtype concept and class inheritance.⁸ Consequently, the formalization of the class hierarchy concept, equally defines a subtype system (on class instance types). In the rest of the thesis, we may use the term “subtype” to describe a subclass, and vice versa.

Since a class hierarchy constructs a finite, non-empty tree which is rooted in the top element `Object`, we immediately achieve an important property.

⁸The correspondance is often referred to as the *inheritance-is-subtyping* property.

Lemma 3.3.4 (The subtype semi-lattice property). The partial subtype order \leq : constructs a *complete lower semi-lattice* on any class hierarchy CH.

We consider the classes `Throwable`, `Runtime` and `Exception` (and some of their subclasses as we will account for in detail in Section 4.4) as an integral part of any class hierarchy CH. This is in accordance with the fact that `java.lang` is an integral part of the Java runtime environment. In practice, however, we allow the following restrictions.

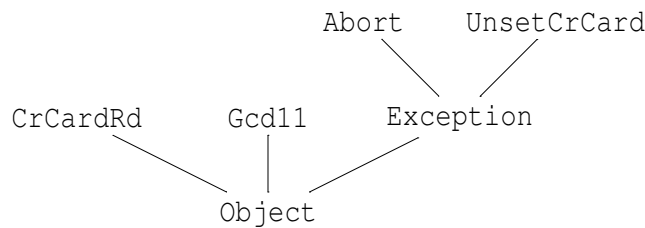
Notation 3.3.5. We generally omit to explicit the subclass relations between `Throwable`, the `Exception` class, and the `Runtime` class (as well as their subclasses) when they are not explicitly subclassed in the class hierarchy.

Finally, we formalize the class hierarchy for our canonical checksum method.

Example 3.3.6 (The checksum class hierarchy formalization). We extend Example 1.3.8 with a formalization of the class hierarchy context for the `cksum()` verification.

$$\text{CH}^{\text{ck}} = \{\text{CrCardRd} \mapsto \text{Object}, \text{Gcd11} \mapsto \text{Object}, \text{Exception} \mapsto \text{Object}, \\ \text{Abort} \mapsto \text{Exception}, \text{UnsetCrCard} \mapsto \text{Exception}\}$$

where “ $_1 \mapsto _2$ ” correspond to the parent map induced by the “ $_1$ extends $_2 \dots$ ” Java clause, *i.e.*, “ $_1 \leq _2$ ”. In this thesis, however, we generally adopt the somewhat counter-intuitive depiction of subclasses above superclasses as follows:



3.4 The Example

We summarize this chapter’s `cksum()` formalizations in Figure 3.8. A detailed description of each formalization step can be found in the Examples 3.1.20, 3.2.5, 3.2.14, and 3.3.6.

```

CP = {1 ↦ methodref(CrCardRd, methsig(getIt, e), int),
      2 ↦ Abort,
      3 ↦ int}
CID = Gcd11

M = ⟨MSIG, RT, EA, CA⟩
MSIG = methsig(cksum, ⟨CrCardRd⟩)
RT = int
EA = ⟨Abort⟩
CA = ⟨MFR, C, ET⟩

MFR = ⟨2, 5⟩
C = aload[1] invokevirtual[1] istore[2] goto[+8] pop
    new[2] athrow ldc_w[3] istore[3] iload[2] iload[3]
    isub istore[4] iload[4] ifle[+10] iload[4] istore[2]
    goto[-16] iload[4] ifne[+6] iload[2] ireturn
    iload[2] istore[4] iload[3] istore[2] iload[4]
    istore[3] goto[-39]
ET = ⟨⟨⟨0, 7⟩, 10, UnsetCrCard⟩⟩

CHck = {CrCardRd ↦ Object, Gcd11 ↦ Object,
        Abort ↦ Exception, UnsetCrCard ↦ Exception}

```

Figure 3.8: Formalizing checksum and its verification context.

Chapter 4

Standard Verification Formalization

In this chapter, we semantically formalize bytecode verification for the considered JVM subset. It extends an earlier formalization given by Rose [47] in several ways. First of all, the notion of type safety has been systematically formalized by a semantical specification of “type assignment compatibility” as an (abstracted) Java type lattice. Second, we consider other JVM features than previously, notably exceptions, array references, and the `null` value. Moreover, we discuss how left-out type-features such as interfaces or multi-dimensional arrays can be added to our model, without invalidating the safety guarantees provided by our verification formalization. Finally, we have added several new instructions, notably those operating with arrays.

In Section 4.1 we discuss how to formalize an abstract model for type safety. In Section 4.2 we discuss our formalization approach to standard verification, and present the semantical judgements for standard verification of a method. In Section 4.3 we semantically specify standard verification of our instruction subset, and in Section 4.4, we discuss the type safety and verification of exceptions. Finally, in Section 4.5, we present a complete standard verification proof-unfolding for our canonical checksum method.

4.1 Analysis and Formalization Strategy

A “type safe program” means that a program which has been statically type verified will run without type errors. The principle was originally formulated by Milner [33]. Its realisation, however, requires the existence of a static type system.

Pure object oriented languages exclusively operate on objects by the invocation of methods which are dynamically dispatched. Type declarations are not available, thus no static type checks are performed.¹ Consequently, all type checks are runtime checks [1]. For more realistic, object oriented languages such as Java, *impure features* have been added, notably: type declarations, primitive types, field variables and compound types. The first of these, type declarations, add on a statically given type system to Java, which permit us to address type safety issues statically. For a class based language such as Java, type safety is related to the close correspondance between the Java subtype concept and the class inheritance principle² as described by many text books [1]. Since

¹“SmallTalk” is an almost pure realization of an object oriented language with no type declarations [18].

²The correspondance is sometimes referred to as the *inheritance-is-subtyping* property.

(instance) method invocation in Java is specified through dynamic dispatch,³ methods become dynamically type-bound, through subsumption,⁴ with their type-environment. Let us consider the static consequences of this in two specific situations. One where a method is invoked on a formal parameter which is dynamically bound to an object, *i.e.*, by a method invocation statement, and one where a method is invoked on an object involved in an assignment statement.

Example 4.1.1 (Type safety for method invocation). Assume that T and S are declared as Java class types.

```
void someMethod (T x) {
    ...
    x.dummy ();
    ...
}
...
someMethod (new S ());
...
```

At compile-time, the formal method parameter x is declared by the class type T in the method declaration, which means that the `dummy ()` method must be declared in either T or one of its super classes, in order for the method to be defined by dynamic dispatch when invoked on x . At runtime, however, x is dynamically bound to an object of class instance type S . In order to prevent the `x.dummy ()` call from failing, we must again verify that S is a subclass of T . With the notation introduced in Section 3.3, the type assignment constraint becomes “ $S \leq T$ ”.

Let us continue with an example of object assignment.

Example 4.1.2 (Type safety for object assignment). Assume that S and T are declared as Java class types.

```
void assignSomeType (S v) {
    T x = v;
    x.dummmmy ();
}
```

When the code is compiled, it is indirectly verified that S is a subtype of T , as the compiler otherwise has to add a downcast check. Furthermore, it is checked that the `dummy ()` method is declared in class T or one of its superclasses (as illustrated in Example 4.1.1). Since x is a local variable, whose assigned values are stored on an *untyped* local variable table location, there is no other way for the bytecode verifier to make direct use of the at source-level declared type. The type constraint which prevent the `x.dummy ()` call from failing, *i.e.*, “ $S \leq T$ ”, is thus ensured by the compiler’s cast-check of S and T , and by a compiler check which ensures that `dummy ()` is declared in either T or a super class.

³Dynamic method dispatch indicates that an (instance) method inheritance is based on the runtime type of the object upon which the method is invoked.

⁴The subsumption rule specifies that an element of a given type also is an element of its supertype. In Abadi and Cardelli’s formulation [1, p.18] : if $a:A$ and $A \leq B$ then $a:B$.

In Example 4.1.1 and 4.1.2 we identified specific static subtype constraints, which can ensure type safety for method invocations. Whether the analysis is general enough to cover all kinds of potential type-unsafe program situations, however, requires a proof which consolidate the general type safety of Java. Such a proof is typically given by the specification of a static and dynamic semantics, which are shown to commute operationally. Since many already have proven that Java/JVM is a type safe language [15, 40], however, and since it is a large and tedious amount of work which is beyond the goal of this thesis, we have decided not to repeat such a proof in this thesis. Instead, we will consider the restatement of Milner’s original type safety formulation [33] in Definition 1.3.1 as our basic hypothesis. Before we continue, however, we formalize an interesting observation on the relationship between statically and dynamically assigned class types.

Observation 4.1.3 (Type safety for dynamic type assignment). Assume that CID_{stat} is a statically declared class (reference) type, and CID_{dyn} is the dynamically assigned (or parameter bound) class instance type. Both declared in the same class hierarchy CH . The following type constraint must hold:

$$(4.1.3a) \quad CID_{dyn} \leq_{:CH} CID_{stat}$$

which reads: a statically declared class type (target type) can be dynamically assigned a class instance type (source) without breaking the type safety, if the dynamically assigned type is a subtype of the statically declared type.

Remark 4.1.4. We notice, that the difference at the JVM level between a statically declared class type, and a dynamically assigned class instance type, is the difference between the way that the types are looked up by the JVM. The statically declared class type, is *directly looked-up in the class file*, as a class reference into the constant pool. The dynamically assigned class instance type, however, is found by *inspection of the instance at the object heap*, in where a class reference to the constant pool will identify the type. At verification time, however, we cannot differentiate between a class instance type and a class reference types, which is why we at some occasions simply write class type to statically describe either.

Finally we notice, that the addition of primitives to Java require that type safety is verified by certain *structural constraints*, e.g., size verifications. Other impure features may be verified by a combination of type assignment constraints and structural checks. Thus, in order to formalize type safety verification, two aspects must be addressed: “well-typedness” (as generally required for object oriented languages), and “well-sizedness” (to address impure object oriented features as earlier discussed) [30, §4.9].

Definition 4.1.5 (General Verification Constraints). The official specification [30, p.142] impose a list of general verification issues.

1. the operand stack must be well-typed and well-sized,
2. the local variable table must be well-typed and well-sized,
3. method invocation must be well-typed,
4. field assignment must be well-typed, and

5. instruction operands must be well-typed.

First, however, we will study how to generalize a type assignment safety-constraint such as “ $s \leq T$ ”, for a local variable x with T as the declared (or static) type and S as the instance (or dynamic) type. The general type constraint concept is called *type assignment compatibility* [30, §2.6.7]. In this thesis, we shall in particular make a study on class types. The concept, however, is also defined for primitive types, interfaces, abstract classes and compound types. During compilation, the local variable x is translated into binary code, which operate on a location in the local variable table. Because local variable locations are untyped, however, their “declared types” are only implicitly given in terms of the type constraints which the bytecode instructions impose on their operands. With this in mind, we reformulate the official specification of type assignment compatibility for an individual local variable location, informally reformulated for our JVM subset [30, §2.6.7].

Definition 4.1.6 (Type assignment compatibility). Consider a situation where a value of dynamic type S (source) is assigned to a location for which the code is statically expecting a value of type T (target).

- if S is the primitive type `int`, it cannot be assigned to a location for a value of class reference type T , or vice versa.
- If T is a class (reference) type, then S must be a subclass or equal to T .
- if S is an array type, and T is not an array type, then T must be of type `Object`.
- If S and T are array types, say $SC []$ and $TC []$, both SC and TC must be of primitive type, *i.e.*, `int`, or,
- SC and TC are classes, and SC is a subclass or equal to TC .

If one of these conditions are satisfied, we say that “ S is assignment compatible with T ”

In order to statically reason over uninitialized or erroneous values,⁵ we extend the formal type sort `Type` by adding a finite number of abstract symbols. First, we have added the abstract type symbols: \top and \perp , to indicate the type of any variable entity, possibly uninitialized. Then we include two additional type symbols: `Null`, to indicate the type of all objects, and $\perp []$, to indicate the type of any array, possibly uninitialized. Let us briefly recall the specification of the `Type` sort in Definition 3.2.2, where it was specified as $\text{Type}_{\text{prim}} \cup \text{Type}_{\text{ob}}$, with Type_{ob} given by $\text{Type}_{\text{cref}} \cup \text{Type}_{\text{aref}}$.

Definition 4.1.7 (The Abstracted Type Sort). We define an extension of the `Type` sort by the addition of abstracted type elements in the following manner.

$$(4.1.7a) \quad \widehat{\text{Type}}_{\text{aref}} = \{\perp []\} \cup \text{Type}_{\text{aref}}$$

$$(4.1.7b) \quad \widehat{\text{Type}}_{\text{ob}} = \{\text{Null}\} \cup \text{Type}_{\text{cref}} \cup \widehat{\text{Type}}_{\text{aref}}$$

⁵Milner was the first to describe analytically, the erroneous state that a variable can be in, by a special value “WRONG” [33].

$$(4.1.7c) \quad \widehat{\text{Type}}_{\perp}^{\top} = \{\perp\} \cup \{\top\} \cup \text{Type}_{\text{prim}} \cup \widehat{\text{Type}}_{\text{ob}}$$

with the ordering as informally specified in Figure 4.1.

Notation 4.1.8. Unless specifically mentioned otherwise, we will from here on use the term *type* to refer to a general element τ in $\widehat{\text{Type}}_{\perp}^{\top}$. Specifically, we let τ_{ob} refer to an element in $\widehat{\text{Type}}_{\text{ob}}$, and τ_{aref} to denote an element in $\widehat{\text{Type}}_{\text{aref}}$.

The Notation 3.2.4 specifies that Type is defined in (3.2.2f) as a restricted set with respect to some class hierarchy CH . According to Notation 3.3.3, CH indicate a finite set of class names, which consequently leads to the following consideration.

Observation 4.1.9. Since any Type sort is a finite set, the abstracted set $\widehat{\text{Type}}_{\perp}^{\top}$ becomes finite.

We begin with a formalization of the type assignment compatibility order relation as an inference system written in natural semantics [14, 22], based on the official, though informal relation description in Definiton 4.1.6. This is in accordance with our general formalization approach. Furthermore, we apply an ML-definition syntax to specify the judgements [34].

Definition 4.1.10 (Type Assignment Compatibility). Type assignment compatibility judgement have the signature:

$$\boxed{\text{ClassHier} \vdash_{\text{bv}} \widehat{\text{Type}}_{\perp}^{\top} \sqsubseteq \widehat{\text{Type}}_{\perp}^{\top}}$$

where ” $\text{CH} \vdash \tau \sqsubseteq \tau'$ ” reads: the type τ' is assignment compatible with the type τ on the abstraction of the type sort Type , restricted by the class hierarchy CH .

$$(4.1.10a) \quad \frac{}{\text{CH} \vdash \tau \sqsubseteq \tau}$$

$$(4.1.10b) \quad \frac{}{\text{CH} \vdash \perp \sqsubseteq \tau}$$

$$(4.1.10c) \quad \frac{}{\text{CH} \vdash \tau \sqsubseteq \top}$$

$$(4.1.10d) \quad \frac{}{\text{CH} \vdash \perp \square \sqsubseteq \text{int} \square}$$

$$(4.1.10e) \quad \frac{}{\text{CH} \vdash \perp \square \sqsubseteq \text{CID} \square}$$

$$(4.1.10f) \quad \frac{}{\text{CH} \vdash \text{Object} \square \sqsubseteq \perp \square}$$

$$(4.1.10g) \quad \frac{}{\text{CH} \vdash \text{Null} \square \sqsubseteq \top}$$

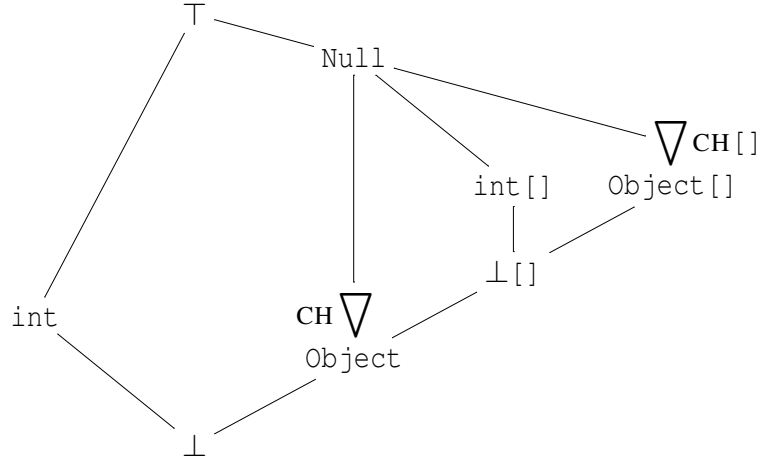


Figure 4.1: A general $\langle \widehat{\text{Type}}_{\perp}^{\top}, \sqsubseteq_{\text{CH}} \rangle$ ordering

$$(4.1.10h) \quad \frac{}{\text{CH} \vdash \text{int} \square \sqsubseteq \text{Null}}$$

$$(4.1.10i) \quad \frac{}{\text{CH} \vdash \text{CID} \square \sqsubseteq \text{Null}}$$

$$(4.1.10j) \quad \frac{}{\text{CH} \vdash \text{CID} \sqsubseteq \text{Null}}$$

$$(4.1.10k) \quad \frac{\text{CH} \vdash \text{CID}_1 \sqsubseteq \text{CID}_2}{\text{CH} \vdash \text{CID}_1 \square \sqsubseteq \text{CID}_2 \square}$$

$$(4.1.10l) \quad \frac{}{\text{CH} \vdash \text{CID}_1 \sqsubseteq \text{CID}_2} \quad \text{where } \text{CID}_2 \leq_{\text{CH}} \text{CID}_1$$

Let us briefly comment on the assignment compatibility formalizations.

- The rule in (4.1.10a) formalizes that a type is assignment compatible with itself.
- The rules in (4.1.10b) to (4.1.10j) formalize the assignment compatibility considerations for the added, abstract type symbols from Definition 4.1.7. We postpone the argumentation for the positioning of the 'Null' type to Remark 4.3.7, and of the $\perp \square$ type to Remark 4.3.12.
- The rule in (4.1.10k) defines assignment compatibility order as *contravariant* in the subtype order on class reference types. A formalization which directly reflects the listed requirements to class types [30, p.22].

- Finally, rule (4.1.10l) formalizes that the assignment compatibility relation is *covariant* in the array element types. A formalization which directly reflects the listed requirements to array reference types [30, p.22].

Based on the formalizations in Definition 4.1.10, we specify the type assignment compatibility relation on the abstracted type set.

Definition 4.1.11 (The Type Assignment Compatibility Relation). The extended type assignment compatibility relation is defined as a binary type order relation on the abstraction of the Type sort, restricted by CH:

$$(4.1.11a) \quad \sqsubseteq_{\text{CH}} \in \widehat{\text{Type}}_{\perp}^{\top} \times \widehat{\text{Type}}_{\perp}^{\top}$$

We write “ $\tau \sqsubseteq_{\text{CH}} \tau'$ ” with the meaning that the (uniquely) associated rule “ $\text{CH} \vdash \tau \sqsubseteq \tau'$ ” can be proven by Definition 4.1.10.

Lemma 4.1.12 (Type compatibility as a partial order). For any class hierarchy CH the relation \sqsubseteq_{CH} is a partial order on the abstracted Type set, restricted by CH.

Proof. From rule (4.1.10a) we immediately have relational reflexivity. From Lemma 3.3.4 follows the transitivity and anti-symmetry from the Java subtype hierarchy, from rule (4.1.10l) these properties are extended to the final, transitive and anti-symmetric organization in Figure 4.1. \square

Let us illustrate a concrete compatibility ordering for our canonical `cksum()` example, given in (1.3.8).

Example 4.1.13 (The checksum compatibility ordering). We illustrate the type compatibility ordering $\langle \widehat{\text{Type}}_{\perp}^{\top}, \sqsubseteq_{\text{CH}^{\text{ck}}} \rangle$ for our canonical checksum example in Figure 4.2, based on the class hierarchy formalization in Example 3.3.6.

According to Observation 4.1.9, any abstracted Type sort is finite. Thus, we can show the following important property.

Proposition 4.1.14 (The extended type lattice). For any class hierarchy CH, the type assignment compatibility order \sqsubseteq_{CH} , constructs a *complete lattice* on the abstracted Type, restricted by CH.

Proof. The ordered set $\langle (\widehat{\text{Type}}_{\perp}^{\top}) \setminus \{\top, \text{Null}\}, \sqsubseteq_{\text{CH}} \rangle$ constructs a finite tree which is rooted in \perp . So, for any $\tau, \tau' \in \widehat{\text{Type}}_{\perp}^{\top} \setminus \{\top, \text{Null}\}$ we have that $\tau \sqcap \tau'$ exists. (Take the smallest subtree completion which contains τ and τ' . The meet will be the root of that subtree.) Moreover we have that $\tau \sqcap \text{Null} = \text{Null} \sqcap \tau = \tau$ is defined for all $\tau \in (\widehat{\text{Type}}_{\perp}^{\top}) \setminus \{\top, \text{int}\}$, and that $\text{int} \sqcap \text{Null} = \text{Null} \sqcap \text{int} = \perp$. Since $\tau \sqcap \tau'$ exists for all $\tau, \tau' \in (\widehat{\text{Type}}_{\perp}^{\top}) \setminus \{\top\}$, we have that the set constructs a *semi-lattice* with the ordering \sqsubseteq_{CH} . Finally we have that $\tau \sqsubseteq \top$ is defined for all $\tau \in (\widehat{\text{Type}}_{\perp}^{\top}) \setminus \{\top\}$. That is \top is a top element for the extended type sort. By lattice theoretical results, we can immediately conclude that $\langle \widehat{\text{Type}}_{\perp}^{\top}, \sqsubseteq_{\text{CH}} \rangle$ constructs a complete lattice [13, Lemma 3.20]. \square

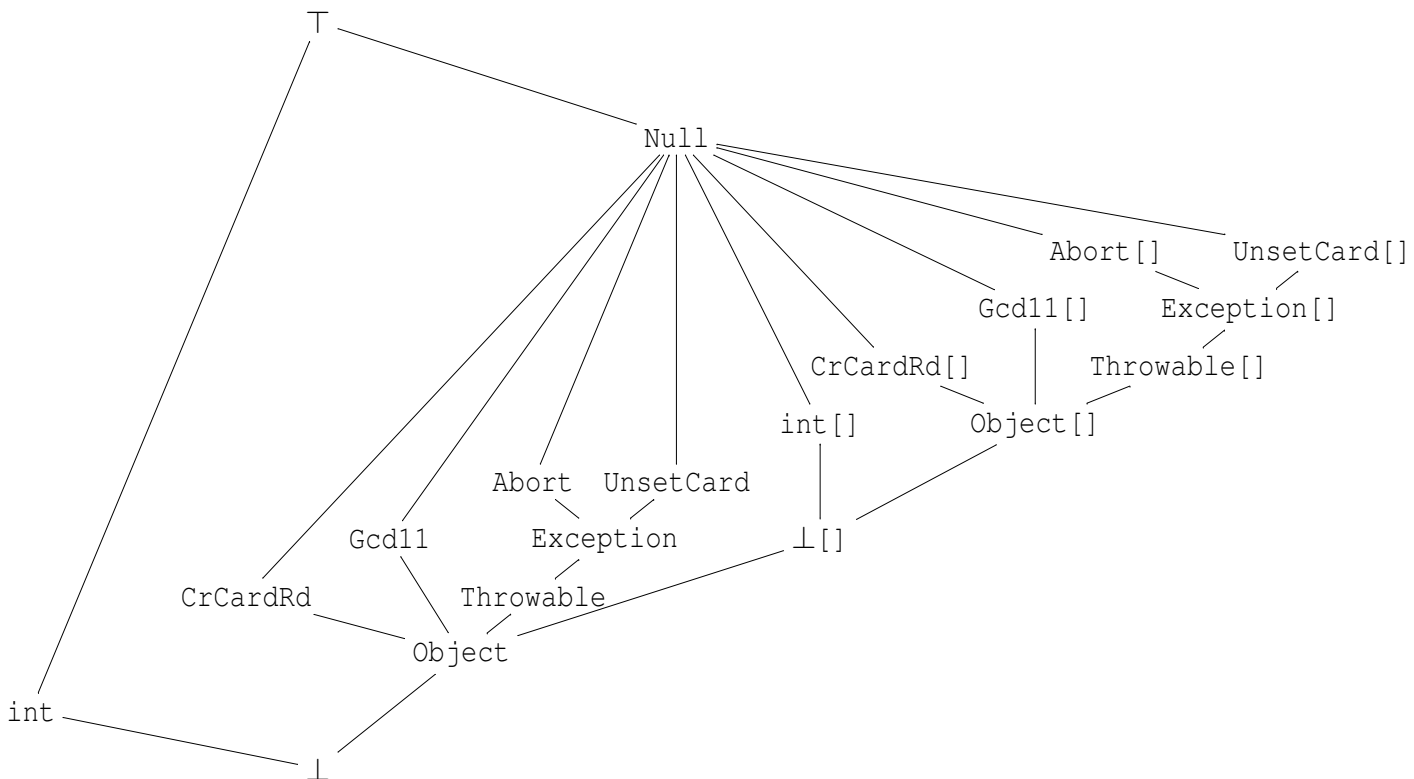


Figure 4.2: The $\langle \widehat{\text{Type}}_{\perp}, \sqsubseteq_{\text{CHek}} \rangle$ example ordering

The general structure of an abstracted type sort $\langle \widehat{\text{Type}}_{\perp}^{\top}, \sqsubseteq_{\text{CH}} \rangle$ is diagrammatically illustrated in Figure 4.1. We notice, that the illustration is *reversed* with respect to the subclass ordering CH (as formalized by rule (4.1.10l)).

Remark 4.1.15 (Multi-dimensional arrays). Notice that adding arrays of arbitrary, bounded dimension is possible without breaking the lattice structure.

Remark 4.1.16 (Interfaces). Interfaces (not considered in this work) are different from classes, in that an interface can extend several other interfaces. This destroys the lattice structure as we know it from classes, which only can extend one class. Presently we have no solution of how interface types should be formally handled by the bytecode verifier.

Observation 4.1.17. The property makes it possible to define a “best defined type” in a type sort $\widehat{\text{Type}}_{\perp}^{\top}$, namely the “meet” type $\tau \sqcap \tau'$, or the “join” type $\tau \sqcup \tau'$ [50, chapter 6]. The first of these defines an abstract Java type which is formally type assignment compatible with either of operands. The property has an interesting consequence for type assignment to the same location. Imagine that at a given method program point, two type constraints τ and τ' are imposed. The property 4.1.14 then ensures the existence of a type solution at that program point.

We recall that bytecode is type safe only when we can statically type-annotate the operand stack elements and the local variable table elements at each program point in the code, such that the instruction’s type constraints are not broken [30, §4.9]. In order to formalize these requirements, we have to formalize the notion of types for a Java frame, *i.e.*, the operand stack and the local variable table, and we have to extend the formal notion of type assignment compatibility to the formalized frame type description.

A method *frame type* is straight forwardly formalized in Definition 4.1.18 as a pair of tuples. The first of these formalizes a general type specification of an operand stack contents, the second one formalizes a general type specification of the contents of a local variable table. The stack type specification is given as an ordered tuple of n elements, where n is the length of the alleged stack for the given method at some program point. The local variable table type specification, abbreviated to “local type”, is an ordered tuple of a fixed number ML of elements, which is the maximal number of allocated variables for the given method.⁶

Definition 4.1.18 (Frame Types).

$$(4.1.18a) \quad \text{FT} \in \text{FrameType}_{\text{MS}, \text{ML}} = \text{StackType}_{\text{MS}} \times \text{LocalType}_{\text{ML}}$$

$$(4.1.18b) \quad \text{ST} \in \text{StackType}_{\text{MS}} = \bigcup_{n=0}^{\text{MS}} \left(\prod_{i=0}^n \tau_i \right), \text{ where } \tau_i \in \widehat{\text{Type}}_{\perp}^{\top}$$

$$(4.1.18c) \quad \text{LT} \in \text{LocalType}_{\text{ML}} = \prod_{i=0}^{\text{ML}} \tau_i, \text{ where } \tau_i \in \widehat{\text{Type}}_{\perp}^{\top}$$

Notation 4.1.19 (Stack types). For a stack type ST of length $k \leq \text{MS}$ which consists of elements τ_i , for $i \in \{0, \dots, k-1\}$, we permit writing $\tau_0 \cdot \tau_1 \cdots \tau_{k-1}$, where the stack top type is given as the rightmost element. Thus, a stack type ‘grows’ towards the right. In accordance with common practice, we may denote a non-empty stack type as $\text{ST}' \cdot \tau$, and an empty stack as ϵ .

⁶The first local variable position must contain a class (instance) type, since we only deal with instance methods.

Notation 4.1.20 (Local types). For a local type LT of length ML which consists of elements $\tau_i, i \in \{0, \dots, ML-1\}$, we permit a local (variable table) type-configuration to be written as $\langle \tau_0, \tau_1, \dots, \tau_{ML-1} \rangle$. We use $LT(i)$ to indicate the type τ_i at the local type index $i \in \{0, \dots, ML-1\}$, and the writing $LT[n \mapsto \tau]$ to denote the local type LT , when updated at index n with the type τ .

The number of stack elements on Java frame operand stacks are formalized as a family of unary operators $|-|_{MS}$, each one indexed with a given, maximal stack length MS . Each of these is recursively defined in the stack type (and always defined, since the size of a stack is bounded).

Definition 4.1.21 (Stack Type Size).

$$(4.1.21a) \quad |-|_{MS} : \text{StackType}_{MS} \rightarrow \mathbf{N}$$

$$(4.1.21b) \quad |e|_{MS} \stackrel{\text{def}}{=} 0$$

$$(4.1.21c) \quad |ST \cdot \tau|_{MS} \stackrel{\text{def}}{=} |ST|_{MS} + 1$$

We extend the definition of the size-operators on stack types to local types. We let the number of allocated variable locations on Java frame local variable tables be formalized as a family of unary operators $|-|_{ML}$, indexed with a given, maximal local variable table size ML .

Definition 4.1.22 (Local Type Size).

$$(4.1.22a) \quad |-|_{ML} : \text{LocalType}_{ML} \rightarrow \mathbf{N}$$

$$(4.1.22b) \quad |\langle \tau_0 \rangle|_1 = 1$$

$$(4.1.22c) \quad |\langle \tau_0, \dots, \tau_{ML-2}, \tau_{ML-1} \rangle|_{ML} \stackrel{\text{def}}{=} |\langle \tau_0, \dots, \tau_{ML-2} \rangle|_{(ML-1)} + 1, \text{ where } ML > 1$$

Remark 4.1.23. Since we only consider instance methods in our formalization, we know that $LT(0)$ is set to the class instance reference type. Thus, we assume that $ML > 0$.

Since all frame types are bounded by their maximal frame sizes, we formalize these as a family of frame type sets $\text{FrameType}_{MS,ML}$, indexed by their maximal frame size constraints $\langle MS, ML \rangle$.

Definition 4.1.24 (Bounded Frame Types).

$$(4.1.24a) \quad \text{FrameType}_{MS,ML} = \{ \langle ST, LT \rangle \mid ST \in \text{StackType}_{MS}, LT \in \text{LocalType}_{ML} \}$$

$$(4.1.24b) \quad \text{StackType}_{MS} = \{ ST \mid ST \in \text{StackType}, |ST| \leq MS \}$$

$$(4.1.24c) \quad \text{LocalType}_{ML} = \{ LT \mid LT \in \text{LocalType}, |LT| = ML \}$$

We continue our formalizations with a specification of assignment compatibility for frame types. Since we see type confusion as a “local” phenomenon which can appear at an individual variable or stack location, we suggest a point-wise extension of the assignment compatibility concept to frame types. Consequently, we define a frame type FT to be assignment compatible with a frame type FT' , provided they are bounded by the same maximal frame type constraints, if all stack type elements of FT are point-wise type assignment compatible with those of FT' , and all local type elements are point-wise type assignment compatible with those of FT' .

In accordance with Definition 4.1.7, however, we begin with an abstraction of the bounded frame types in order to be able to reason over uninitialized or erroneous Java frames.

Definition 4.1.25 (The Frame Type Sort).

$$(4.1.25a) \quad (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top} = \{\perp_{\text{MS,ML}}\} \cup \{\top_{\text{MS,ML}}\} \cup \text{FrameType}_{\text{MS,ML}}$$

where for each $\langle \text{MS,ML} \rangle$, all frame types $\text{FT} \in (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$ are frame type compatible with $\perp_{\text{MS,ML}}$, and $\top_{\text{MS,ML}}$ frame type compatible with FT .

Notation 4.1.26. By abuse of notation, we will also let FT refer to an element in the type sort $(\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$, whenever the meaning is clear from the context.

Remark 4.1.27 (The StackType). According to Observation 3.1.10, no local variable instruction may access the local variable table without accessing the stack. Hence, it is *not observable* to differentiate between when a stack is not well defined or when the entire frame is not well defined. Statically this means that we do not need to differentiate between the abstract $\langle \perp_{\text{MS}}, \text{LT} \rangle$ type (*e.g.*, when two stack type constraints are incompatible) and $\perp_{\text{MS,ML}}$ (*e.g.*, when the complete frame type is undefined). Consequently, *no* undefined stack type \perp_{MS} has been added to the $\text{StackType}_{\text{MS}}$ formalization in Definition 4.1.30.

We can now specify frame type assignment compatibility by an inference system, written in *natural semantics* [14, 22]. We will use the judgment sort notation of the Standard ML definition [34].

Definition 4.1.28 (Frame Type Assignment Compatibility). Frame type assignment compatibility judgements have the signature:

$$\boxed{\text{ClassHier} \vdash_{\text{bv}} (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top} \sqsubseteq_{\text{MS,ML}} (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}}$$

where “ $\text{CH} \vdash \text{FT} \sqsubseteq_{\text{MS,ML}} \text{FT}'$ ” reads: the frame type FT' is frame type assignment compatible with the frame type FT over the same abstracted, and bounded frame type sort $(\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$, where the individual frame type constituents are elements of the same extended type sort Type , restricted by CH .

$$(4.1.28a) \quad \frac{}{\text{CH} \vdash \text{FT} \sqsubseteq_{\text{MS,ML}} \text{FT}}$$

$$(4.1.28b) \quad \frac{\text{CH} \vdash \text{FT}_1 \sqsubseteq_{\text{MS,ML}} \text{FT}_2 \quad \text{CH} \vdash \text{FT}_2 \sqsubseteq_{\text{MS,ML}} \text{FT}_3}{\text{CH} \vdash \text{FT}_1 \sqsubseteq_{\text{MS,ML}} \text{FT}_3}$$

$$(4.1.28c) \quad \frac{}{\text{CH} \vdash \perp \sqsubseteq_{\text{MS,ML}} \text{FT}}$$

$$(4.1.28d) \quad \frac{}{\text{CH} \vdash \text{FT} \sqsubseteq_{\text{MS,ML}} \top}$$

$$(4.1.28e) \quad \frac{\text{CH} \vdash \text{ST}_1 \sqsubseteq_{\text{MS}} \text{ST}_2 \quad \text{CH} \vdash \text{LT}_1 \sqsubseteq_{\text{ML}} \text{LT}_2}{\text{CH} \vdash \text{FT}_1 \sqsubseteq_{\text{MS,ML}} \text{FT}_2}$$

(4.1.28e.i) where $FT_1 = \langle ST_1, LT_1 \rangle$

(4.1.28e.ii) $FT_2 = \langle ST_2, LT_2 \rangle$

For our convenience we add two special cases.

$$(4.1.28f) \quad \frac{CH \vdash FT_1 \sqsubseteq_{MS,ML} FT_2}{CH \vdash FT_1 =_{MS,ML} FT_2} \quad FT_1 = FT_2$$

$$(4.1.28g) \quad \frac{CH \vdash FT_1 \sqsubseteq_{MS,ML} FT_2}{CH \vdash FT_1 \sqsubset_{MS,ML} FT_2} \quad FT_1 \neq FT_2$$

Since the side conditions are mutually exclusive, the following holds.

Observation 4.1.29. (4.1.28f) and (4.1.28g) are mutually exclusive.

The assignment compatibility concept has specifically been formalized for stack types and local types.

Definition 4.1.30 (Stack And Local Type Assignment Compatibility). The stack type assignment compatibility judgement has the signature:

$$\boxed{\text{ClassHier} \vdash_{bv} \text{StackType}_{MS} \sqsubseteq_{MS} \text{StackType}_{MS}}$$

where “ $CH \vdash ST \sqsubseteq_{MS} ST'$ ” reads: the stack type ST' is stack type assignment compatible with the stack type ST on the MS bounded stack type sort StackType_{MS} , where the individual stack type constituents are elements of the same extended type sort Type , restricted by CH .

$$(4.1.30a) \quad \frac{}{CH \vdash \epsilon \sqsubseteq_{MS} \epsilon}$$

$$(4.1.30b) \quad \frac{CH \vdash ST_1' \sqsubseteq_{MS} ST_2' \quad CH \vdash \tau_{11(|ST_1|-1)} \sqsubseteq \tau_{21(|ST_2|-1)}}{CH \vdash ST_1 \sqsubseteq_{MS} ST_2}$$

$$(4.1.30b.i) \quad \text{where } |ST_1| \leq MS$$

$$(4.1.30b.ii) \quad |ST_1| = |ST_2|$$

$$(4.1.30b.iii) \quad ST_1 = ST_1' \cdot \tau_{11(|ST_1|-1)}$$

$$(4.1.30b.iv) \quad ST_2 = ST_2' \cdot \tau_{21(|ST_2|-1)}$$

$$(4.1.30b.v)$$

The local type assignment compatibility judgement has the signature:

$$\boxed{\text{ClassHier} \vdash_{bv} \text{LocalType}_{ML} \sqsubseteq_{ML} \text{LocalType}_{ML}}$$

where “ $CH \vdash LT \sqsubseteq_{ML} LT'$ ” reads: the local type LT' is local type assignment compatible with the local type LT on the ML bounded local type sort LocalType_{ML} , where the individual local type constituents are elements of the same extended type sort Type , restricted by CH .

$$(4.1.30c) \quad \frac{}{\text{CH} \vdash \epsilon \sqsubseteq_0 \epsilon}$$

$$(4.1.30d) \quad \frac{\text{CH} \vdash \text{LT}_1' \sqsubseteq_{\text{ML}-1} \text{LT}_2' \quad \text{CH} \vdash \tau_{12(\text{ML}-1)} \sqsubseteq \tau_{22(\text{ML}-1)}}{\text{CH} \vdash \text{LT}_1 \sqsubseteq_{\text{ML}} \text{LT}_2}$$

$$(4.1.30d.i) \quad \text{where } |\text{LT}_1| = \text{ML}$$

$$(4.1.30d.ii) \quad |\text{LT}_1| = |\text{LT}_2|$$

$$(4.1.30d.iii) \quad \text{LT}_1 = \langle \text{LT}_1', \tau_{12(\text{ML}-1)} \rangle$$

$$(4.1.30d.iv) \quad \text{LT}_2 = \langle \text{LT}_2', \tau_{22(\text{ML}-1)} \rangle$$

where $\tau_{i,j,k}$ is given by either

- $\tau_{i,1,k} \in \widehat{\text{Type}}_{\perp}^{\top}$ for each $i \in \{1,2\}$, $k \in \{0, \dots, (|\text{ST}_i| - 1)\}$, or
- $\tau_{i,2,k} \in \widehat{\text{Type}}_{\perp}^{\top}$ for each $i \in \{1,2\}$, $k \in \{0, \dots, (\text{ML} - 1)\}$, respectively.

Remark 4.1.31. Definition 4.1.28 and Definition 4.1.30 together realize that frame type assignment compatibility constructs a *point-wise extension* of $\widehat{\text{Type}}_{\perp}^{\top}$ to $\text{StackType}_{\text{MS}}$ and $\text{LocalType}_{\text{ML}}$, and whence to $(\text{FrameType}_{\text{MS,ML}})$.

Finally, we shall briefly comment on how we have chosen our abstract frame type assignment compatibility, stack type compatibility, and our local type assignment compatibility formalizations.

- The rule in (4.1.28a) and (4.1.28b) formalizes the reflexive and transitive nature of frame type assignment compatibility. Even though this also follows from the point-wise way that the formalization of frame type assignment compatibility has been defined for ordinary type assignment compatibility on $\text{FrameType}_{\text{MS}}^{\text{ML}}$, we have decided to add these properties explicitly, since the point-wise definition does not hold on extended frame type sorts, $(\text{FrameType}_{\text{MS}}^{\text{ML}})_{\perp}^{\top}$.
- In our formalization of stack type assignment compatibility in (4.1.30b) we have followed the official specification which prescribes that at every program point, we must know the *stack height* and every element on it [30, p.144]. Thus we require that *only frame types where the stack type components have the same size* can be assignment comparable. At first glance, one could object that a method which consists of the following code, should be allowed to verify. Even when the two possible branches from label 3 (the ifne conditional) will produce two stack types of different length at label 6, the next instruction will only access the top of the stack, which in both cases contains the required integer. Let us consider a pseudo dataflow analysis on a pseudo instruction sequence.

```

0 iconst_1      int
1 ...           int.int
2 ifne         int.int

```

```

3 ...          int
4 ...          int
5 iconst_2    int
6 ireturn     int.int  int  -->  proposed merge: int

```

It is tempting to suggest that $\text{int} \sqsubseteq_{\text{MS}} \text{int} \cdot \text{int}$ with int as the “best defined” stack type by which the two stack branches are comparable. However, we would then lose track of the stack size produced by the other branch, and that way we could very well cause a violation of the method’s stack size constraint (MS) at run-time.

Based on the formalizations in Definition 4.1.28, we specify the frame type assignment compatibility relation on the abstracted frame type set.

Definition 4.1.32 (The Frame Type Assignment Compatibility Relation). The extended frame type assignment compatibility relation is defined as a binary type order relation on the abstraction of the $\text{FrameType}_{\text{MS,ML}}$ sort, an abstraction of the Type sort, restricted by CH:

$$(4.1.32a) \quad \sqsubseteq_{\text{MS,ML}}^{\text{CH}} \in (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top} \times (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$$

We write “ $\text{FT} \sqsubseteq_{\text{MS,ML}}^{\text{CH}} \text{FT}'$ ” with the meaning that the (uniquely) associated rule “ $\text{CH} \vdash \text{FT} \sqsubseteq_{\text{MS,ML}} \text{FT}'$ ” can be proven by Definition 4.1.28.

Definition 4.1.33 (The Stack and Local Type Assignment Compatibility Relation). In parallel with Definition 4.1.32, we introduce “ $\sqsubseteq_{\text{MS}}^{\text{CH}} \in \text{StackType}_{\text{MS}} \times \text{StackType}_{\text{MS}}$ ” as a binary type order relation, and “ $\sqsubseteq_{\text{ML}}^{\text{CH}} \in \text{LocalType}_{\text{ML}} \times \text{LocalType}_{\text{ML}}$ ” as a binary type order relation, based on their operational specifications in Definition 4.1.30.

Lemma 4.1.34 (Frame type compatibility as a partial order). For arbitrary frame constraints $\langle \text{MS}, \text{ML} \rangle$, the relation $\sqsubseteq_{\text{MS,ML}}^{\text{CH}}$ is a partial order relation on the frame type set $(\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$ which is abstracted from TYPE, restricted by CH.

Proof. Easy by observing that the frame type (carrier) set $(\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$ is constructed from the Type (carrier) set using just operations from Definition 2.3.2 and equipped with the implied partial orderings. \square

Observation 4.1.35. According to Observation 4.1.9, an abstracted Type sort is finite. In parallel with the proof of Lemma 4.1.34, we immediately have that $(\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$ is finite.

Proposition 4.1.36 (The frame type lattice property). The frame type assignment compatibility order $\sqsubseteq_{\text{MS,ML}}^{\text{CH}}$ constructs a *complete lattice* on $(\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$ for any class hierarchy CH.

Proof. Follows immediately from Proposition 4.1.14 and Proposition 2.3.5. \square

From the frame type lattice property, we obtain the following frame type existence statement, where $\sqcap_{\text{MS,ML}}$ and $\sqcup_{\text{MS,ML}}$ are given the usual meaning [50, chapter 6].

Corollary 4.1.37 (The “meet” and “join” frame type existence). *For all frame types $FT_1, FT_2 \in (\text{FrameType}_{MS,ML})_{\perp}^{\top}$, there exists a unique “meet” frame type $FT_1 \sqcap_{MS,ML} FT_2$ such that the frame type $(FT_1 \sqcap_{MS,ML} FT_2) \sqsubseteq_{MS,ML} FT_1$ and $(FT_1 \sqcap_{MS,ML} FT_2) \sqsubseteq_{MS,ML} FT_2$, and there exists a unique “join” frame type $FT_1 \sqcup_{MS,ML} FT_2$ such that $(FT_1 \sqcup_{MS,ML} FT_2) \sqsubseteq_{MS,ML} FT_1$, and such that the frame type $(FT_1 \sqcup_{MS,ML} FT_2) \sqsubseteq_{MS,ML} FT_2$.*

The corollary extends (pointwise) to the following important lemma.

Lemma 4.1.38 (Algebraic properties). We have that $\langle \langle \sqsubseteq_{MS,ML}, (\text{FrameType}_{MS,ML})_{\perp}^{\top} \rangle, \sqcap_{MS,ML} \rangle$ and $\langle \langle \sqsubseteq_{MS,ML}, (\text{FrameType}_{MS,ML})_{\perp}^{\top} \rangle, \sqcup_{MS,ML} \rangle$ are commutative and associative.

Proof. Follows immediately from the fact that $\langle \sqsubseteq_{MS,ML}, (\text{FrameType}_{MS,ML})_{\perp}^{\top} \rangle$ constructs a complete lattice, *i.e.*, all of its subsets have a well-defined “top” element, and a well-defined “bottom” element, which is *independent* of the order in which the elements are arranged. \square

4.2 Bytecode Verification

“Standard bytecode verification” is officially specified as a dataflow algorithm which exploits that bytecode is explicitly typed to decide whether it is type safe by the ability to statically construct a set of Java frame type annotations which is *proportional* to the code size. In dataflow terms *a solution* (or typing) to the method’s constraint set. In Observation 1.3.3, we noticed that the mere existence of some “appropriate” static frame type typing which satisfies the method’s constraint set at all program points, guarantee that the method code is well-typed, *i.e.*, type safe.

In this thesis, we formalize standard bytecode verification seen as a type checker, which given a frame type typing for the method, checks whether it satisfies the method’s constraint set. Thus, we do not intend to formalize a verification strategy which actual iterates a frame type solution, *e.g.*, as an abstract interpretation over an abstract JVM type domain.⁷ We notice, however, that abstract interpretation translates to solving “chaotic equations” in a complete lattice. In our context, this would mean that the order in which a method code’s (abstract) static type constraints (“equations”) are solved during an abstract (type safety) interpretation, is indifferent (“chaotic”) with respect to the existence of a (well-typed) solution [11] (as the abstracted frame type domain constitutes a complete lattice, *cf.*, Property 4.1.36).

Observation 4.2.1. With our standard verification specification-approach, *the order* in which we perform the type checks, *i.e.*, “unfold” the proofs, is *indifferent* with respect to the checking result.

Before we specify standard bytecode verification, however, we formalize an arbitrary method typing as a (finite) map from, or an *assignment* of, a method’s program point set to the set of abstracted frame types, but where *none* of the formal values can be \top nor \perp .

Definition 4.2.2 (Frame Type Typing). Let a method with code C have its frame size bounded by MS, ML . A frame type typing for that method is given by the algebraic sort specification:

$$(4.2.2a) \quad \text{FTA} \in \text{FrameTypeAssign}_{MS,ML} = \text{PPoint}_C \rightarrow \text{FrameType}_{MS,ML}$$

⁷The iteration of a solution to the abstract interpretation is formally a fixed point iteration.

The initial method invocation frame type is well-defined and formalizes to the ϵ -abstraction for the empty operand stack, followed by an abstraction of the current self-reference type CID_0 at local variable table location 0, and the abstraction of its, say k , formal parameter types in the k subsequent local variable table locations.

Definition 4.2.3 (Invocation Frame Type). Let the maximal frame size be given by MS, ML .

$$(4.2.3a) \quad FT_0 \in (\text{FrameType}_{MS,ML})_{\perp}^{\top}$$

$$(4.2.3b) \quad FT_0 = \langle \epsilon, \langle CID_0, T_1, \dots, T_k, \perp_{k+1}, \dots, \perp_{ML-1} \rangle \rangle; 0 \leq k < ML$$

We illustrate this with a formalization of the invocation frame for our canonical `checksum()`.

Example 4.2.4 (The Checksum Invocation Frame Type).

$$FT_0^{ck} \in (\text{FrameType}_{2,5})_{\perp}^{\top}$$

$$FT_0^{ck} = \langle \epsilon, \langle \text{Gcd11}, \text{CrCardRd}, \perp, \perp, \perp \rangle \rangle$$

We now proceed with the actual bytecode verification formalization. Since this formalization is supposed to serve as the basis for defining lightweight bytecode verification, we will not specify it as a constraint system from which a satisfactory frame type assignment has to be deduced. We observe, that for any satisfactory bytecode verification, there exists a frame type assignment FTA , defined on the verified method's set of program points, on the extended frame type set FrameTypeAssign . The frame type assignment is by definition the “meet” of all frame types which at any program point satisfies the set of type safety constraints that must hold for the alleged frame type assignment, regardless of the evaluation order.

With this approach in mind, we specify bytecode verification as a collection of logical judgments which formalizes the set of type safety constraints that must hold at every program point, one judgment for every program point. Specifically, we choose natural semantics as our specifying formalism [22]. Hence, a successful bytecode verification of some method will correspond to the construction of a (logical) proof for the frame type assignment and the verified method, which is *finite in the number of exhaustive proof unfoldings*. Since bytecode verification is formalized as a big step semantics, we will specify the individual verification formalization steps in a top-down manner. The actual judgment system which formally defines bytecode verification, is specified by Definition 4.2.7 through Definition 4.4.20.

Definition 4.2.5 (The Formal Verification Assumptions). We briefly list the major verification assumptions for our formalizations.

- The class file and class hierarchy must be well-formed in accordance with the specifications in Chapter 3 (a requirement which corresponds to the official expectations that a structural format check of the bytecode is performed prior to bytecode verification [30, §4]).
- The initial method invocation frame type, formalized in Definition 4.2.3, is considered as well-typed prior to bytecode verification of that method (which includes that it is called with the correct number and type of arguments). This relies on the assumption that the individual method verifications of some class file, only takes place, if the method invocation call can be correctly verified. An assumption which corresponds to the officially described class file verification procedure [30, § 4].

- We assume that any instruction I in the method code C is uniquely identified by its opcode position PP , relative to the code. In particular, we assume that the program points are given in ascending numeric order during the instruction verification formalizations in Section 4.3.
- We assume that the FTA map, which is specified in the formalizations, is defined in all program points $\text{Dom}(C)$ of the associated method.
- Finally, we refer to Definition 4.3.4 for a listing of general, judgment notational conventions.

We introduce the verification contexts Δ , called the *method verification context*, and Ω , called the *code verification context*. The former specifies method-specific compile-time information, the latter specifies the complete compile-time information for a method's standard verification (that is the class file and hierarchy context, the method-specific context, and a frame type typing).

Definition 4.2.6 (The Verification Contexts).

$$(4.2.6a) \quad \Delta \in \text{MethContext}_{\text{MS,ML}} = \text{ReturnType} \times \text{MaxFrame}_{\text{MS,ML}} \times \text{ExcAtt} \times \text{ExcTable}$$

$$(4.2.6b) \quad \Omega \in \text{CodeContext}_{\text{MS,ML}} = \text{StdContext} \times \text{MethContext}_{\text{MS,ML}} \times \text{FrameTypeAssign}_{\text{MS,ML}}$$

Before we specify the method standard verification judgement, let us briefly comment on the initial and final verification requirements.

Initial constraints The initial constraints on the lightweight verifier are

- the initial expected frame-type is FT_0 ,
- the initial accumulated set of program points is \emptyset .

The initial expected frame type FT_0 is specified as the method's invocation frame type in the side condition (4.2.7a.ix). The initial set of accumulated program points, however, is directly indicated in the rightmost premise.

Final constraints The final constraints on the lightweight verifier are

- the “accumulating-condition” in (4.2.7a.x)

is the ultimate requirement for a successful method verification. It states that the accumulated set of program points for standard verified instructions, must comprise all program points of the method.

Definition 4.2.7 (Method Standard Verification). A method verification judgement has the signature

$$\boxed{\text{StdContext} \vdash_{\text{bv}} \text{Method}, \text{FrameTypeAssign}}$$

where “ $\Gamma \vdash_{\text{bv}} M, \text{FTA}$ ” reads: the method M bytecode verifies with the frame type assignment FTA in the verification context Γ .

$$(4.2.7a) \quad \frac{\text{CH} \vdash \text{FTA}(\emptyset) \sqsubseteq \text{FT}_0 \quad \Omega \vdash \emptyset, C, \emptyset \xRightarrow{\text{tsafe}} \text{PPS}}{\Gamma \vdash_{\text{bv}} M, \text{FTA}}$$

(4.2.7a.i)	where	$\Omega = \langle \Gamma, \Delta, \text{FTA} \rangle$
(4.2.7a.ii)		$\Gamma = \langle \langle \text{CP}, \text{CID} \rangle, \text{CH} \rangle$
(4.2.7a.iii)		$\Delta = \langle \text{RT}, \text{MFR}, \text{EA}, \text{ET} \rangle$
(4.2.7a.iv)		$\text{M} = \langle \text{MSIG}, \text{RT}, \text{EA}, \text{CA} \rangle$
(4.2.7a.v)		$\text{CA} = \langle \text{MFR}, \text{C}, \text{ET} \rangle$
(4.2.7a.vi)		$\text{MFR} = \langle \text{MS}, \text{ML} \rangle$
(4.2.7a.vii)		$\text{MSIG} = \text{methsig}(\text{ID}, \text{T}_1, \dots, \text{T}_k)$
(4.2.7a.viii)		$0 \leq k < \text{ML}$
(4.2.7a.ix)		$\text{FT}_0 = \langle \epsilon, \langle \text{CID}, \text{T}_1, \dots, \text{T}_k, \perp_{k+1}, \dots, \perp_{\text{ML}-1} \rangle \rangle$
(4.2.7a.x)		$\text{PPS} = \text{PPS}_C$

The code sequence verification is specified by two rules. For a code sequence of one instruction, and for a code sequence of several instructions. We notice, that a well-defined method in Java contains at least one instruction. Thus, the sequence rule set is well-defined for all legal method code components. Then we observe that the individual instructions in a method code component are associated with distinct program points. Thus, the side-condition (4.2.8b.iii) in Rule 4.2.8b, will successively accumulate the program points of all verified instructions, by a stepwise verification of the associated instructions.

Definition 4.2.8 (Instruction Sequence Verification). An instruction sequence verification judgment has the signature

$$\boxed{\text{CodeContext} \vdash_{\text{bv}} \text{PPoint}, \text{Code}, \text{PPoints} \xRightarrow{\text{tsafe}} \text{PPoints}}$$

where “ $\Omega \vdash \text{PP}, \text{CS}, \text{PPS} \xRightarrow{\text{tsafe}} \text{PPS}'$ ” reads : the code sequence CS, starting at program point PP with a set of prior, verified program points PPS, is verified in the code verification context Ω , with the accumulated set of verified program points PPS' .

$$(4.2.8a) \quad \frac{\Omega \vdash \text{PP}, \text{I} \xrightarrow{\text{tsafe}} \text{PP}', \top}{\Omega \vdash \text{PP}, \text{I}, \text{PPS} \xRightarrow{\text{tsafe}} \text{PPS}'}$$

$$(4.2.8a.i) \quad \text{where } \text{PPS}' = \text{PPS} \cup \{\text{PP}\}$$

$$(4.2.8b) \quad \frac{\begin{array}{l} \Omega \vdash \text{PP} : \text{I}_1 \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}, \\ \text{CH} \vdash \text{FTA}(\text{PP}') \sqsubseteq \text{FT}_{\text{PP}'}^{\text{PP}}, \end{array}}{\Omega \vdash \text{PP}', \text{I}_2 \cdot \text{CS}, \text{PPS}' \xRightarrow{\text{tsafe}} \text{PPS}''} \frac{}{\Omega \vdash \text{PP}, \text{I}_1 \cdot \text{I}_2 \cdot \text{CS}, \text{PPS} \xRightarrow{\text{tsafe}} \text{PPS}''}$$

$$\begin{aligned}
(4.2.8b.i) \quad & \text{where } \Omega = \langle \Gamma, -, \text{FTA} \rangle \\
(4.2.8b.ii) \quad & \Gamma = \langle -, \text{CH} \rangle \\
(4.2.8b.iii) \quad & \text{PPS}' = \text{PPS} \cup \{\text{PP}\}
\end{aligned}$$

Remark 4.2.9. We notice that Judgment 4.2.8 implies a slight abuse of notation, as PP cannot be a program point of an empty code segment. As the last program point is ignored by our specification, however, we side-step this and by abuse of notation assume that $\text{PP} \in \text{PPoints}$.

4.3 Instruction Verification

Instruction verification is given as a formalization of the instruction-specific type constraints in Definition 4.1.5. Based on the observation that the set of verification constraints are virtually the same for each of the instruction groups specified in Section 3.1, we intend to ease the readability by only presenting one inference rule per instruction group. Thus, we introduce a syntactically sugared inference rule notation, which also will be referred to as a *family of inference rules*. The rule notation is realized as one, shared inference for each of the instruction groups which are specified in Section 3.1. The side condition syntax is enriched by a *verification table*, which schematically presents those instruction-specific side conditions which differs within each instruction-group.

Notation 4.3.1 (The Verification Table Notation). The verification table is defined as *syntactic sugar* for a set of side conditions which are specific for each individual instruction. Let us consider a verification table which covers two kinds of safety conditions, given by column X and Y.

I	X	Y
<i>i</i>	<i>x</i>	<i>y</i>

Each table line is interpreted as follows: if the instruction being verified is *i*, the conditions “ $X = x$ ” and “ $Y = y$ ” must hold in particular, for the instruction *i*. We notice that an “unfolding”, *i.e.*, a syntactical “de-sugaring” of such a table, eventually splits the inference rule into several, instruction-specific inference rules. For example, a “syntactical unfolding” of the presented verification table would cause the inference rule for instruction *i* to have “ $X = x$ ” and “ $Y = y$ ” added to its side conditions.

Before we proceed, we must introduce a new frame type concept.

Definition 4.3.2 (The Expected Frame Type). By an *expected frame type* in PP' , we understand the frame type which is imposed from the instruction at the previous program point PP by application of its compile-time semantics on the frame type constraint at PP. Specifically we define

- the expected frame type in 0 to be the invocation frame type FT_0 ,
- the expected frame type after a non-fall through instruction to be \top .

An expected frame type is denoted by $FT_{PP'}^{PP}$, whenever PP is a preceding program point to PP'

We begin by a specification of the (commonly shared) instruction-group judgment signature.

Definition 4.3.3 (The Instruction Verification Signature). An instruction verification judgment has the signature

$$\boxed{\text{CodeContext} \vdash_{\text{bv}} \text{PPoint} : \text{Ins} \xrightarrow{\text{tsafe}} \text{PPoint}, \text{FrameType}}$$

where “ $\Omega \vdash PP : I \xrightarrow{\text{tsafe}} PP', FT_{PP'}^{PP}$ ” reads: the instruction I , at the program point PP , is verifiable in the code verification context Ω with the successor program point PP' and the expected frame type $FT_{PP'}^{PP}$.

We list the specification assumptions which our formalizations must satisfy.

Notation 4.3.4 (Assumptions and notational conventions).

- The formal assumptions in Definition 4.2.5.
- The side conditions of inference rules which are specified by Definition 4.3.3 are generally organized into three groups. The uppermost side condition group *specifies the context* by pattern matching. Then follows the side conditions which assure that the code constraints are *well-typed*, and finally we list those constraints which assures that the *well-sized* constraints are satisfied, in accordance with Definition 4.1.5. In separate verification tables, we finally list the instructions and their instruction-specific characteristics (if any).
- We mark sort elements which are unbound by the inference rule by $_$. In (4.3.6a.iv), *e.g.*, we have replaced the class file context $\langle \text{CP}, \text{CID} \rangle$ in the verification context Γ with $_$, as it is not referenced.
- We generally use the notation PP' to signify the first byte position after the verified instruction at PP . As noticed in Remark 4.3.5, PP' does not always signify a program point in the method being verified. When it does, however, the frame type assignment $\text{FTA}(PP')$, and the expected frame type $FT_{PP'}^{PP}$, are well-defined. Specifically, we often use PP'' to signify a jump target.
- Furthermore, we have omitted the subscripts 'MS' and 'ML' from the \sqsubseteq symbol, in order to ease the readability to the extent where they are not obvious from the context.
- We use the notation $FT_{PP'}^{PP}$, for arbitrary program points PP and PP' , to denote the frame type, imposed by application of the compile-time semantics on the instruction at PP on the program point PP'
- In order to ease readability we allow the writing $\text{FTA}(PP)$ in judgments and inference rules, even though it is a syntactical abuse of notation. We notice that a correct but more cumbersome approach, would be either to introduce function application syntactically, or to deal with it in inference rule side conditions.

Remark 4.3.5. As we verify the last method instruction at the program point PP, the successive position PP', does not indicate a byte position of an instruction in the method. Since there are no type constraints imposed at PP', we permit, *by abuse of notation* that $PP' \in \text{PPoint}$.

Stack instructions have briefly been described in Definition 3.1.8. As indicated by the name, these instructions purely operate on the operand stack, as specified in Figure 3.1. This means that these instructions expect a certain stack structure in order to execute to a well-defined run-time stack state. Stack instruction bytecode verification translates these expectations to the following static type constraints. When primitive (integer) typed stack elements are manipulated, these stack requirements are straight forwardly formalized into static constraints as depicted in the verification table for the stack verification judgement. The instructions dup and pop, however, manipulate *any* kind of stack top element. In this sense they are type *polymorphic*.⁸ At compile-time, “any kind of stack element” translates into a stack type element of type $\tau \in \text{Type}$. In Remark 4.3.7, we argue that the null value is special in that it can be of any object reference type (instances or arrays) at run-time. Since there are no polymorphic types available in Java we shall do with the special, abstract type Null, which was added to our abstract type domain in Definition 4.1.7, with the ordering depicted in Figure 4.1. The addition helps, *e.g.*, in formalizing the type constraints imposed by the aconst_null instruction. The instruction-specific type-constraints are formalized in the verification table to the stack instruction judgement.

Definition 4.3.6 (Stack Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$\begin{aligned}
 (4.3.6a) \quad & \frac{}{\Omega \vdash \text{PP} : \text{I} \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}} \\
 (4.3.6a.i) \quad & \text{where } \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle \\
 (4.3.6a.ii) \quad & \text{FTA}(\text{PP}) = \langle \text{ST}, \text{LT} \rangle \\
 (4.3.6a.iii) \quad & \text{FT}_{\text{PP}'}^{\text{PP}} = \langle \text{ST}', \text{LT} \rangle \\
 (4.3.6a.iv) \quad & \Gamma = \langle _, \text{CH} \rangle \\
 (4.3.6a.v) \quad & \Delta = \langle _, \langle \text{MS}, _ \rangle, _, _ \rangle \\
 (4.3.6a.vi) \quad & \text{PP}' = \text{PP} + 1 \\
 (4.3.6a.vii) \quad & |\text{ST}'| < \text{MS}
 \end{aligned}$$

I	ST	ST'
iconst_0	ST	ST · int
iconst_1	ST	ST · int
aconst_null	ST	ST · Null
dup	ST ₁ · τ	ST ₁ · τ · τ
pop	ST ₁ · τ	ST ₁
iadd	ST ₁ · int · int	ST ₁ · int
isub	ST ₁ · int · int	ST ₁ · int

⁸A “polymorphic type” is used to denote a type variable which can be assigned a non-polymorphic (ground) type. For example used to specify an input-output function for a program.

Discussion 4.3.7 (The `Null` type). As illuded to above, the `null` value in Java is special in that an expression, assigned to `null`, may be of any (object) type at runtime. In order to give a static type interpretation of the effect of the `aconst_null` instruction, we would like to indicate that the runtime type can be any object type. In static type terms, this means that we would like to define a “most general object type” with which any potential runtime object-type is compatible. Since there are no polymorphic types available in Java, we have introduced a special `Null` type to our Java Type sort in Definition 4.1.7, as illustrated in Figure 4.1.

The *Local variable instructions* have briefly been specified in Definition 3.1.9. They are characterized by their impact on the local variable table, as specified by the run-time execution scheme in Figure 3.2. We notice, however, that the local variable instructions all operate on the stack, and thus expect a certain stack structure in order to execute to a well-defined run-time state. Since local variable instructions only concern moving an element from the stack to the variable table or vice versa, we must require strict *compile-time type-identity* between the element type which is expected to be moved, and the element type expected to be received at either stack or local variable table. The instruction-specific type safety requirements described in Figure 3.2 have been straight forwardly formalized in the local variable judgement verification table.

Definition 4.3.8 (Local Variable Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$\begin{aligned}
 (4.3.8a) \quad & \frac{}{\Omega \vdash PP : I \xrightarrow{\text{tsafe}} PP', FT_{PP'}^{PP}} \\
 (4.3.8a.i) \quad & \text{where } \Omega = \langle \Gamma, \Delta, FTA \rangle \\
 (4.3.8a.ii) \quad & FTA(PP) = \langle ST, LT \rangle \\
 (4.3.8a.iii) \quad & FT_{PP'}^{PP} = \langle ST', LT' \rangle \\
 (4.3.8a.iv) \quad & \Gamma = \langle -, CH \rangle \\
 (4.3.8a.v) \quad & \Delta = \langle -, \langle MS, ML \rangle, -, - \rangle \\
 (4.3.8a.vi) \quad & PP' = PP + 2 \\
 (4.3.8a.vii) \quad & |ST'| < MS \\
 (4.3.8a.viii) \quad & n < ML
 \end{aligned}$$

I	ST	ST'	LT'
(4.3.8a.ix) <code>istore[n]</code>	$ST' \cdot \text{int}$	ST'	$LT[n \mapsto \text{int}]$
<code>astore[n]</code>	$ST' \cdot \tau_{\text{ob}}$	ST'	$LT[n \mapsto \tau_{\text{ob}}]$
<code>iload[n]</code>	ST	$ST \cdot \text{int}$ where $LT(n) = \text{int}$	LT
<code>aload[n]</code>	ST	$ST \cdot \tau_{\text{ob}}$ where $LT(n) = \tau_{\text{ob}}$	LT

Before we continue the instruction verification formalizations, we need to formalize exceptions which can be launched from the same program point.

Definition 4.3.9 (Launched Exception Formalization). We formalize the exceptions which can be launched from a given program point as an unspecified exception name sort.

$$(4.3.9a) \quad ES \in \text{Excs} = \text{ClassIdent}^*$$

Even though exception verification is postponed to Section 4.4, the order in which exceptions are verified is *independent* of each other, with the following consequence.

Notation 4.3.10 (Exception Representation). Exceptions which can be thrown at the same program point may be written

$$E_1 \cdot E_2 \dots E_n$$

where the listing order is indifferent.

Array instructions have already been specified in Definition 3.1.11. As indicated by the name, these instructions primarily operate between the heap and the stack, as specified by the run-time execution scheme of Figure 3.3. We observe that any array operation in our verification framework is only reflected indirectly by the resulting stack operations, since the heap is not part of the verified structures. For the most part, they therefore expect a certain stack structure in order to execute to a well-defined run-time state. Notice in particular how the `arraylength` instruction poses no requirements on the type of its stack elements, here formalized by τ , which by convention is an element in the $\widehat{\text{Type}}$ sort. As for local variable instructions, we must require strict *static type-identity* between the element type which is expected to be stored or loaded, and the element type which has actually been fetched. The stack type expectations have been formalized in the verification table of the array verification judgement. The table furthermore lists a set of miscellaneous instruction-specific evaluation conditions, including the special exceptions which may be cast.

An array rule has two premises in our formalization. The leftmost premise assures that the the frame type assignment FTA is assignment compatible with the successor frame type in the next program point at PP' . The rightmost premise, verifies the instruction specific exceptions which may be thrown by the instruction. The listing of these exceptions is given in a second verification table, whereas the actual discussion and formalization of exception verification is postponed to Section 4.4.

Definition 4.3.11 (Array Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.11a) \quad \frac{\Theta \vdash \text{PP}, \text{ES}, \text{ET}}{\Omega \vdash \text{PP} : \text{I} \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{\text{PP}'}}^{\text{PP}'}$$

$$\begin{aligned} (4.3.11a.i) \quad & \text{where } \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle \\ (4.3.11a.ii) \quad & \Theta = \langle \Omega, \text{LT}, \text{false} \rangle \\ (4.3.11a.iii) \quad & \text{FTA}(\text{PP}) = \langle \text{ST}, \text{LT} \rangle \\ (4.3.11a.iv) \quad & \text{FT}_{\text{PP}'}^{\text{PP}} = \langle \text{ST}', \text{LT} \rangle \\ (4.3.11a.v) \quad & \Delta = \langle -, \langle \text{MS}, - \rangle, -, \text{ET} \rangle \\ (4.3.11a.vi) \quad & |\text{ST}'| < \text{MS} \end{aligned}$$

I	ST	ST'	condition	PP'
iaload	$ST_1 \cdot \text{int}[] \cdot \text{int}$	$ST_1 \cdot \text{int}$		PP + 1
aaload	$ST_1 \cdot \text{CID}[] \cdot \text{int}$	$ST_1 \cdot \text{CID}$		PP + 1
iastore	$ST' \cdot \text{int}[] \cdot \text{int} \cdot \text{int}$	ST'		PP + 1
aastore	$ST' \cdot \text{CID}[] \cdot \text{int} \cdot \text{CID}'$	ST'	$\text{CID}' \leq \text{CID}$	PP + 1
newarray_int	$ST_1 \cdot \text{int}$	$ST_1 \cdot \text{int}[]$		PP + 2
anewarray[n]	$ST_1 \cdot \text{int}$	$ST_1 \cdot \text{CID}[]$	$\text{CP}[n] = \text{CID}$	PP + 3
arraylength	$ST_1 \cdot \tau[]$	$ST_1 \cdot \text{int}$		PP + 1

I	ES
iaload	NullPointerException · ArrayIndexOutOfBoundsException
aaload	NullPointerException · ArrayIndexOutOfBoundsException
iastore	NullPointerException · ArrayIndexOutOfBoundsException
aastore	NullPointerException · ArrayIndexOutOfBoundsException
	...ArrayStoreException
newarray_int	NegativeArraySizeException
anewarray[n]	NegativeArraySizeException
arraylength	NullPointerException

Discussion 4.3.12 (An array of “any type”). We notice that the `arraylength` instruction is specified to count the number of elements in an array, regardless of their type, and regardless of whether the array is initialized or not. Specifically, we make this observation *observable* for our static type verification, by the addition of the special abstract type $\perp[]$, as it has been in Definition 4.1.7.

The following example illustrates the point: the following code allocates two different arrays and uses shared code to access the length:

```
Method void main(java.lang.String[])
  0 aload_0
  1 arraylength
  2 ifne 12
  5 iconst_5
  6 newarray int
  8 astore_1
  9 goto 18
12 bipush 6
14 anewarray class #2 <Class java.lang.Object>
17 astore_1
18 getstatic #3 <Field java.io.PrintStream out>
21 aload_1
22 arraylength
23 invokevirtual #4 <Method void println(int)>
26 return
```

This is rejected by the verifier with the error

```
Exception in thread "main" java.lang.VerifyError:
  (class: BotArray, method: test signature: ([Ljava/lang/String;)V)
  Expecting to find array on stack
```

in spite of the fact that the code is type-safe.

Constant pool instructions have been briefly specified in Definition 3.1.13. These instructions primarily operate between the constant pool and the stack, as specified by the run-time execution scheme of Figure 3.4. For the most part, they expect a certain stack structure in order to execute to a well-defined run-time state.

In order to ease readability, we will structure the formalization by three different judgments, each one for a partition of the constant pool instructions. The first is specified in Definition 4.3.13, the second in Definition 4.3.16, whereas the third specification is given by Definition 4.3.17. The first kinds are called *simple access* instructions, since these instructions only retrieve values specified by the Type sort from the constant pool. (The *anewarray* instruction is particular in that it is both an array instruction and a constant pool instruction. In this representation, however, we have decided to let the grouping of array instructions take precedence over other groupings.) The second kinds are described as *field access* instructions, since these instructions only retrieve values of the FieldType sort from the constant pool. Finally, the last kind is called *method invocation* instructions, since these instructions only retrieve method descriptors of the MethodType sort from the constant pool.

All of the three constant pool rules are verified by two premises. The leftmost premise assures that the the frame type assignment FTA is assignment compatible with the successor frame type in the next program point at PP' . The rightmost premise, verifies the instruction-specific exceptions which may be thrown by the instruction. The listing of these exceptions is given in a second verification table, whereas the actual discussion and formalization of exception verification is postponed to Section 4.4.

Definition 4.3.13 (Simple Access, Constant Pool Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.13a) \quad \frac{\Theta \vdash PP, ES, ET}{\Omega \vdash PP : I \xrightarrow{\text{tsafe}} PP', FT_{PP'}^{PP}}$$

(4.3.13a.i) where $\Omega = \langle \Gamma, \Delta, FTA \rangle$

(4.3.13a.ii) $\Theta = \langle \Omega, LT, \text{false} \rangle$

(4.3.13a.iii) $FTA(PP) = \langle ST, LT \rangle$

(4.3.13a.iv) $FT_{PP'}^{PP} = \langle ST', LT \rangle$

(4.3.13a.v) $\Gamma = \langle \langle CP, - \rangle, CH \rangle$

(4.3.13a.vi) $\Delta = \langle -, \langle MS, - \rangle, -, ET \rangle$

(4.3.13a.vii) $PP' = PP + 3$

(4.3.13a.viii) $|ST'| < MS$

(4.3.13a.ix)

I	ST	ST'	CP[n]	ES
checkcast[n]	$ST_1 \cdot \tau_{ob}$	$ST_1 \cdot \tau_{ob}$	τ'_{ob}	ClassCastException
ldc_w[n]	ST	$ST \cdot \text{int}$	int	ϵ
new[n]	ST	$ST \cdot \text{CID}$	CID	ϵ

Remark 4.3.14 (The constant pool index). Notice how we verify the constant pool index *indirectly* by assuming that the type of the referenced constant pool item is well-defined.

Remark 4.3.15 (The checkcast instruction). Notice how the checkcast instruction primarily is a *runtime check*, which guarantees that a type cast of a stack element is performed “upwards” to an item of object type in the constant pool. (*e.g.*, from a subclass to a super class, subarray to a super array/Object). Thus, it is generally unpredictable to perform this check statically, and therefore only the individual stack and constant pool item types, as well as the stack structure is verified.

Field valued constant pool instructions are specified as for simple valued verification, yet with an additional type constraint on the field argument, added as a premise. A constraint which corresponds to the usual type assignment compatibility requirement for field assignment, where τ by convention is an element in the $\widehat{\text{Type}}$ sort. Furthermore, an additional side condition verifies that the class where the field is applied, *i.e.*, CID, is a subclass of the class where the field is declared, *i.e.*, CID' in our formalization. A constraint which, by the *method inheritance property*,⁹ ensures that the field has a value at runtime, since the runtime type of the applied field always will be identical to, or a subclass to, its compile-time type.

Definition 4.3.16 (Field Access, Constant Pool Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.16a) \quad \frac{\text{CH} \vdash \text{T} \sqsubseteq \tau \quad \Theta \vdash \text{PP}, \text{ES}, \text{ET}}{\Omega \vdash \text{PP} : \text{I} \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{\text{PP}'}}^{\text{tsafe}}$$

$$(4.3.16a.i) \quad \text{where} \quad \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle$$

$$(4.3.16a.ii) \quad \Theta = \langle \Omega, \text{LT}, \text{false} \rangle$$

$$(4.3.16a.iii) \quad \text{FTA}(\text{PP}) = \langle \text{ST}, \text{LT} \rangle$$

$$(4.3.16a.iv) \quad \text{FT}_{\text{PP}'}^{\text{PP}} = \langle \text{ST}', \text{LT} \rangle$$

$$(4.3.16a.v) \quad \Gamma = \langle \langle \text{CP}, - \rangle, \text{CH} \rangle$$

$$(4.3.16a.vi) \quad \Delta = \langle -, -, -, \text{ET} \rangle$$

$$(4.3.16a.vii) \quad \text{PP}' = \text{PP} + 3$$

$$(4.3.16a.viii) \quad \text{CID} \leq_{:\text{CH}} \text{CID}'$$

$$(4.3.16a.ix) \quad \begin{array}{|c|c|c|c|c|} \hline \text{I} & \text{ST} & \text{ST}' & \text{CP}[n] & \text{ES} \\ \hline \text{putfield}[n] & \text{ST}_1 \cdot \text{CID} \cdot \tau & \text{ST}_1 & \text{fieldref}(\text{CID}', -, \text{T}) & \text{NullPointerException} \\ \hline \text{getfield}[n] & \text{ST}_1 \cdot \text{CID} & \text{ST}_1 \cdot \text{T} & \text{fieldref}(\text{CID}', -, \text{T}) & \text{NullPointerException} \\ \hline \end{array}$$

v

Method valued constant pool instructions are basically specified as for field valued instruction verification, yet with an additional set of method argument type constraint added as j premises

⁹An application of a *field* can be seen as another way to *make a “getter” method call* in order to fetch the field value.

to the verification rule in (4.3.17a). These constraints correspond to the usual type assignment compatibility requirements on method calls, where τ by convention is an element in $\widehat{\text{Type}}$. In order to enhance the readability, we have furthermore listed the instruction-specific side conditions for these instructions in two additional verification tables. As for the verification of fields, an additional side condition verifies that the class within which the method is called, *i.e.*, CID , is a subclass of the class where the method is declared, *i.e.*, CID' in our formalization. A constraint which, by the *method inheritance property* ensures that the field has a value at runtime, since the runtime type of an applied method always is identical to, or a subclass to, the methods compile-time type. Furthermore we notice, that since *all* listed exceptions which can be thrown from an `invokevirtual` program point, we have listed those exceptions as a constraint set in (4.3.17a.ix) instead of a list, in accordance with the conventions described in Notation 4.3.10 (knowing that such a set will be finite for any concrete method verification).

Definition 4.3.17 (Method Invocation, Constant Pool Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.17a) \quad \frac{\begin{array}{c} \text{CH} \vdash T_1 \sqsubseteq \tau_1 \\ \vdots \\ \text{CH} \vdash T_j \sqsubseteq \tau_j \\ \Theta \vdash \text{PP}, \text{ES}, \text{ET} \end{array}}{\Omega \vdash \text{PP} : \text{I} \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{\text{PP}'}}}$$

$$(4.3.17a.i) \quad \text{where} \quad \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle$$

$$(4.3.17a.ii) \quad \Theta = \langle \Omega, \text{LT}, \text{false} \rangle$$

$$(4.3.17a.iii) \quad \text{FTA}(\text{PP}) = \langle \text{ST}, \text{LT} \rangle$$

$$(4.3.17a.iv) \quad \text{FT}_{\text{PP}'}^{\text{PP}} = \langle \text{ST}', \text{LT} \rangle$$

$$(4.3.17a.v) \quad \Gamma = \langle \langle \text{CP}, _ \rangle, \text{CH} \rangle$$

$$(4.3.17a.vi) \quad \Delta = \langle _, _, _, \text{ET} \rangle$$

$$(4.3.17a.vii) \quad \text{PP}' = \text{PP} + 3$$

$$(4.3.17a.viii) \quad \begin{array}{|c|c|c|} \hline \text{ST} & \text{ST}' & \text{CP}(n) \\ \hline \text{ST}_1 \cdot \text{CID} \cdot \tau_1 \dots \tau_j & \text{ST}_1 & \text{methref}(\text{CID}', \text{methsig}(_, T_1, \dots, T_j), \text{void}) \\ \hline \text{ST}_1 \cdot \text{CID} \cdot \tau_1 \dots \tau_j & \text{ST}_1 \cdot T & \text{methodref}(\text{CID}', \text{methsig}(_, T_1, \dots, T_j), T) \\ \hline \end{array}$$

$$(4.3.17a.ix) \quad \begin{array}{|c|c|c|} \hline \text{I} & \text{condition} & \text{ES} \\ \hline \text{invokevirtual}[n] & \text{CID} \leq_{\text{CH}} \text{CID}' & \{E \mid E \leq_{\text{CH}} \text{Throwable}\} \\ \hline \end{array}$$

The exceptions which may be rethrown by `invokevirtual` (that includes *all* exceptions which our instruction subset can throw), are specified as subclasses of `Throwable`. In order to clarify our formal exception verification treatment (indicated by the lowest of the premises), we only consider those exceptions which our instruction subset may throw.

Definition 4.3.18 (A Representative Exception Subset). Subclasses of `Throwable` which are covered by our subset consist of the following parts.

- The subclasses of `RuntimeException` which are listed in Definition 4.3.6 through Definition 4.3.25 (which, by definition, are part of the class hierarchy CH).

$$\text{ES}_{\text{RunExc}} \stackrel{\text{def}}{=} \{\text{NegativeArraySizeException}, \text{ArrayStoreException}, \\ \text{ArrayIndexOutOfBoundsException}, \text{ClassCastException}, \\ \text{NullPointerException}\}$$

- The subclasses of `Exception`, given within the class hierarchy CH. (That is all user-defined exceptions in a given method.)

Remark 4.3.19 (Omitted exceptions). In order to clarify our formal presentation, we have omitted `Error` and any of its subclasses as well as any of the system-defined subclasses of `Exception`.

The next instruction group to formalize is the *conditional code jump* group, which was briefly described in Definition 3.1.15. As described by Figure 3.5, the runtime effect of one of those instructions is a binary branched execution control transfer, depending on a condition value. Statically, this behaviour is verified by imposing the *same type constraints* at both (potential) jump targets. These requirements have been formalized as side conditions to the rule in (4.3.20a), where PP' and PP'' signifies the two potential jump targets, and ST' the commonly expected stack type. The additional, instruction-specific requirements on the type of the jump condition, have been systematically listed in the adherent verification table. Finally we notice, that the branch rule is specified by two premises, which assures that the frame type assignment is frame type assignment compatible with the expected frame type in the successor program point PP' as well as the expected frame type at the jump target PP'' .

Definition 4.3.20 (Branch Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.20a) \quad \frac{\text{CH} \vdash \text{FTA}(PP'') \sqsubseteq \text{FT}_{PP'}^{PP}}{\Omega \vdash \text{PP} : \text{I} \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{PP'}^{PP}}$$

$$(4.3.20a.i) \quad \text{where} \quad \Omega = \langle \Gamma, -, \text{FTA} \rangle$$

$$(4.3.20a.ii) \quad \text{FTA}(PP) = \langle \text{ST}, \text{LT} \rangle$$

$$(4.3.20a.iii) \quad \text{FT}_{PP'}^{PP} = \langle \text{ST}', \text{LT} \rangle$$

$$(4.3.20a.iv) \quad \Gamma = \langle -, \text{CH} \rangle$$

$$(4.3.20a.v) \quad \text{PP}' = \text{PP} + 3$$

$$(4.3.20a.vi) \quad \text{PP}'' = \text{PP} + \mathfrak{n}$$

I	ST
ifne[n]	$\text{ST}' \cdot \text{int}$
ifle[n]	$\text{ST}' \cdot \text{int}$
ifnull[n]	$\text{ST}' \cdot \tau_{\text{ob}}$

(4.3.20a.vii)

Remark 4.3.21 (Jump targets). Notice, that the verification of whether a code jump target is well-defined in some method, only is specified *indirectly*; we simply require that the method’s frame type assignment FTA, is well-defined in the jump target PP'' .

In the remainder of this section, we present a bytecode verification specification of what in assembler technology is known as *non-fall through instructions*. For our subset, these are: goto[n], athrow, return, ireturn, and areturn.

The first of these is the *unconditional code jump* instruction goto[n], which was briefly described in Definition 3.1.16. As specified by Figure 3.6, the runtime effect of a goto execution is an immediate control transfer to the jump target PP'' which is indicated by the argument (byte) n, *regardless* of the frame state. Statically, we formalize this by requiring that the frame type at the jump target is *identical* to the frame type at the jump source (which here is $FTA(PP)$.)

Finally we notice that goto imposes no type constraints on the subsequent program point (if any).

Definition 4.3.22 (Goto Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.22a) \quad \frac{CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP}}{\Omega \vdash PP : \text{goto} \xrightarrow{\text{tsafe}} PP', \top}$$

$$(4.3.22a.i) \quad \text{where } \Omega = \langle \Gamma, -, FTA \rangle$$

$$(4.3.22a.ii) \quad FT_{PP''}^{PP} = FTA(PP)$$

$$(4.3.22a.iii) \quad \Gamma = \langle -, CH \rangle$$

$$(4.3.22a.iv) \quad PP' = PP + 3$$

$$(4.3.22a.v) \quad PP'' = PP + n$$

Remark 4.3.23. We notice, that goto does not impose any type constraints at the source jump in PP . Consequently, any kind of frame type may be imposed at the jump target in PP'' , as the source constraint may be \top or \perp .

The next group of non-fall through instructions encompasses the *abrupt instructions*. The first of these, the athrow instruction, was briefly described in Definition 3.1.17 and Figure 3.6. It causes an immediate execution control transfer, since an exception is thrown onto the operand stack. We notice, that since the exception is dynamically checked by the Java runtime system, the stack does not need to be statically type verified.

The transfer is either *local*, *i.e.*, to the location of the first matching handler, or to the *invoker of the method*, *i.e.*, when the exception is rethrown by the method. The actual athrow type verification is formalized by three rules which differ in the kind of exceptions verification strategy they apply. We notice, that the actual exception verification, given by the premises, is treated in Section 4.4.

The rule in (4.3.24a) is for any thrown exception which is not a user-defined Exception. The rule in (4.3.24b) is for any thrown exception which is a *user-defined* Exception, and finally, the rule in (4.3.24c) is for any thrown exception which is a *system-defined* RuntimeException. Because of the hierarchy structure between these built-in classes [8], we have that the rules are mutually exclusive. The reason for the rule separation into “user-defined Exceptions” is because we have

to check that these exceptions are declared by the throwing method(’s attribute table EA), as this decides which local variables should be type checked. The reason for the rule separation into “system-defined RunTime exceptions”, or for the rule separation into “other exceptions” which are not an Exception (or subclass hereoff), is that these exceptions do *not* have to be declared by the user. We notice, that in the side conditions, the different rules “propagate” a “toggle” set to true or false, which is packed into the environment variable Θ in the exception verification premises. The toggle value decides the which of the described verification strategy to apply.

Finally we notice, that since the frame is popped after the execution control is transferred, athrow impose no type constraints on the following program point (if any).

Definition 4.3.24 (Throw Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.24a) \quad \frac{\Theta \vdash PP, CID_E, ET}{\Omega \vdash PP : \text{athrow} \xrightarrow{\text{tsafe}} PP', \top}$$

$$(4.3.24a.i) \quad \text{where } \Omega = \langle \Gamma, \Delta, FTA \rangle$$

$$(4.3.24a.ii) \quad \Theta = \langle \Omega, LT, \text{false} \rangle$$

$$(4.3.24a.iii) \quad FTA(PP) = \langle CID_E \cdot -, LT \rangle$$

$$(4.3.24a.iv) \quad \Gamma = \langle -, CH \rangle$$

$$(4.3.24a.v) \quad \Delta = \langle -, -, -, ET \rangle$$

$$(4.3.24a.vi) \quad CID_E \leq_{CH} \text{Throwable}$$

$$(4.3.24a.vii) \quad CID_E \not\leq_{CH} \text{Exception}$$

$$(4.3.24a.viii) \quad PP' = PP + 1$$

$$(4.3.24b) \quad \frac{\Theta \vdash PP, CID_E, ET}{\Omega \vdash PP : \text{athrow} \xrightarrow{\text{tsafe}} PP', \top}$$

$$(4.3.24b.i) \quad \text{where } \Omega = \langle \Gamma, \Delta, FTA \rangle$$

$$(4.3.24b.ii) \quad \Theta = \langle \Omega, LT, \text{true} \rangle$$

$$(4.3.24b.iii) \quad FTA(PP) = \langle CID_E \cdot -, LT \rangle$$

$$(4.3.24b.iv) \quad \Gamma = \langle -, CH \rangle$$

$$(4.3.24b.v) \quad \Delta = \langle -, -, -, ET \rangle$$

$$(4.3.24b.vi) \quad CID_E \leq_{CH} \text{Exception}$$

$$(4.3.24b.vii) \quad CID_E \not\leq_{CH} \text{RuntimeException}$$

$$(4.3.24b.viii) \quad PP' = PP + 1$$

$$(4.3.24c) \quad \frac{\Theta \vdash PP, CID_E, ET}{\Omega \vdash PP : \text{athrow} \xrightarrow{\text{tsafe}} PP', \top}$$

$$\begin{array}{ll}
(4.3.24c.i) & \text{where } \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle \\
(4.3.24c.ii) & \Theta = \langle \Omega, \text{LT}, \text{false} \rangle \\
(4.3.24c.iii) & \text{FTA}(\text{PP}) = \langle \text{CID}_E \cdot -, \text{LT} \rangle \\
(4.3.24c.iv) & \Gamma = \langle -, \text{CH} \rangle \\
(4.3.24c.v) & \Delta = \langle -, -, \text{EA}, \text{ET} \rangle \\
(4.3.24c.vi) & \text{CID}_E \leq_{\text{CH}} \text{RunTimeException} \\
(4.3.24c.vii) & \text{PP}' = \text{PP} + 1
\end{array}$$

where the premises are specified in Definition 4.4.11.

The last group of *abrupt instructions* are the return instructions which have been briefly described in Definition 3.1.18. As specified in Figure 3.6, the runtime effect is an immediate execution control transfer back to the method invoker. Any value which the method must return, however, must be present on the stack top prior to a return instruction execution. The latter situation is verified by an assignment compatibility check of the statically returned type on the stack, and the method's declared return type in the constant pool. Finally we notice, that since the frame is discarded after the execution control is returned to the method invoker, these instruction imposes no type constraints on the following program point (if any).

Definition 4.3.25 (Return Instruction Verification). Take the judgment signature to be as in Definition 4.3.3.

$$(4.3.25a) \quad \frac{}{\Omega \vdash \text{PP} : \text{return} \xrightarrow{\text{tsafe}} \text{PP}', \top}$$

$$\begin{array}{ll}
(4.3.25a.i) & \text{where } \Omega = \langle -, \Delta, \text{FTA} \rangle \\
(4.3.25a.ii) & \Delta = \langle \text{void}, -, -, - \rangle \\
(4.3.25a.iii) & \text{PP}' = \text{PP} + 1
\end{array}$$

$$(4.3.25b) \quad \frac{}{\Omega \vdash \text{PP} : \text{ireturn} \xrightarrow{\text{tsafe}} \text{PP}', \top}$$

$$\begin{array}{ll}
(4.3.25b.i) & \text{where } \Omega = \langle -, \Delta, \text{FTA} \rangle \\
(4.3.25b.ii) & \text{FTA}(\text{PP}) = \langle \text{ST} \cdot \text{int}, - \rangle \\
(4.3.25b.iii) & \Delta = \langle \text{int}, -, -, - \rangle \\
(4.3.25b.iv) & \text{PP}' = \text{PP} + 1
\end{array}$$

$$(4.3.25c) \quad \frac{\text{CH} \vdash \text{T}_{\text{ob}} \sqsubseteq \tau_{\text{ob}}}{\Omega \vdash \text{PP} : \text{areturn} \xrightarrow{\text{tsafe}} \text{PP}', \top}$$

$$\begin{aligned}
(4.3.25c.i) \quad & \text{where } \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle \\
(4.3.25c.ii) \quad & \text{FTA}(\text{PP}) = \langle \text{ST} \cdot \tau_{\text{ob}}, - \rangle \\
(4.3.25c.iii) \quad & \Gamma = \langle -, \text{CH} \rangle \\
(4.3.25c.iv) \quad & \Delta = \langle \text{T}_{\text{ob}}, -, -, - \rangle \\
(4.3.25c.v) \quad & \text{PP}' = \text{PP} + 1
\end{aligned}$$

From our semantical verification descriptions for non-fall through instructions, we summarize an interesting property.

Observation 4.3.26 (Non-fall through instruction constraints). All expected frame type constraints after non-fall through instructions are \top within our formalization, *i.e.*, they impose no type constraints on any subsequent program points.

4.4 Exception Verification

In this section we will discuss how type safety verification apply to exceptions, as well as how to decide for an appropriate exception verification strategy. Finally, we present the formalization of our verification strategy.

Discussion 4.4.1 (Description of an exception.) The semantical impact of an exception can be seen as though it transforms the instruction which raised it into a *jump instruction*, or an *abrupt instruction*, depending on whether the exception is caught by one of the enclosing method's exception handlers, or whether it is rethrown. Two questions arise in the attempt to specify bytecode verification of exceptions. First, *how do we interpret the exception recovery pattern semantics of checked and unchecked exceptions in our formalization?* Second, *to what extent is it possible to assure these recovery patterns statically*, that is by an analysis based on the exception's compile-time types? The first question is best answered by a discussion on what we understand by exception type safety; the second, by investigating an appropriate verification strategy. As a first approach to type safety for exceptions, we consider an example of a type confusion situation which can appear at runtime.

Example 4.4.2 (Exception type confusion). A variation of the `cksum()` method code is shown in figure 4.3. This variant is unsafe, however, because an exception, raised by the `invokevirtual` instruction, is caught by the handler at program point 10, but violates $\text{FTA}(2) \sqsupseteq \text{FTA}(10)$. (For details on the exception table, we refer to Figure 4.6.) The frame type violation appears since the local variable type at program point 10, at the index 2 contains `Abort` which isn't assignment compatible with `CrCardRd`, which is contained at the index 2 of the local variable type at program point 2. In formal terms: $\text{Abort} \not\sqsupseteq \text{CrCardRd}$. We notice, that an `Abort` reference at the stack-top will be dynamically checked by the Java runtime system, whereas the local variable table, which is an inherent scope from the local variable state of the source jump, must be verified statically if the type error should be prevented.

PP	FTA _{pp} .ST	FTA _{pp} .LT (this·ccnum·x·y·z)	I
0	ε	Gcd11·CrCardRd·⊥·⊥·⊥	aload[1]
2	CrCardRd	Gcd11·CrCardRd·⊥·⊥·⊥	invokevirtual[1]
5	int	Gcd11·CrCardRd·⊥·⊥·⊥	istore[2]
7	ε	Gcd11·CrCardRd·int·⊥·⊥	goto[+8]
10	UnsetCrCard	Gcd11·Abort·⊥·⊥·⊥	pop
11	ε	Gcd11·Abort·⊥·⊥·⊥	aload[1]
13	Abort	Gcd11·Abort·⊥·⊥·⊥	athrow
⋮			

Figure 4.3: An unsafe cksum() variant.

In order to prevent such situations, we proceed by a discussion of how to ensure exception type safety for an exception handle.

Discussion 4.4.3 (Exception Type Safety). Exception safety guarantees has to provide the same kind of type safety precautions as for jump instructions and abrupt instructions.

Jump behaviour: when a local handle catches the raised exception. We require the same assignment compatibility, type safety constraint at the catching handle, as for any other jump target.

Abrupt behaviour: when the raised exception is uncaught by the method's exception table. If the exception is checked, we require that the exception is declared¹⁰ by the method's exception attribute.

Based on Example 4.4.2 and this discussion it seems crucial, in an exception handler situation, to establish the same kind of type safety as at usual jump targets, *i.e.*, *type compatibility at the handling location*. Even though exceptions are bytecode verified in practice, little has been written to formally specify after which principles it is done. Thus, we have tried to account for what we consider as necessary and reasonable formalization strategy.

If we consider what we have called an exception's compile-time type, it is formally possible, with the present formalizations, to give a static type safety description of exceptions and their handlers, specified by the exception table. First we recall, that a class type of a handle in the exception table (in the class file), remains the same at compile-time and runtime. Thus, it is only the instance-type of the exception which may differ from the compile-time type. In Definition 4.1.3, we have formally specified how an instance type may differ at run-time and compile-time, a definition which permit us to categorize the exception an the handle's catch capabilities by three compile-time situations.

Definition 4.4.4 (Compile-time Catch Situations). Assume that an exception has a compile-time type which can be formalized by CID_E. Furthermore, assume that an enclosing method exception handler, which cover the localtion where the exception can be raised, is formalized by $\langle PP_1, PP_2, PP'', CID \rangle$.

¹⁰A declared exception is either listed by the exception attribute, or a subclass to a listed class.

1. Only when $CID_E \leq CID$, we can be *sure* that the handle will catch the exception at runtime. We call the situation for a *definite catch*.
2. Assume that $CID \leq CID_E$, *i.e.*, CID_E is a super class type of CID at compile-time. According to Definition 4.1.3, we *may or may not* have that the exception is caught by the handler at runtime. We call the situation for a *potential catch*.
3. The exception will never be handled when the program point from where the exception may be raised is out of range, *or* if both $CID_E \not\leq CID$ and $CID \not\leq CID_E$ (as the exception type and handle's class type remain incomparable at runtime, *cf.*, Definition 4.1.3). We call the situation for a *no catch*.

Finally, we must discuss type safety of exceptions which imposes an abrupt behaviour. Since a situation like this appear when an exception is uncaught by the exception table, two things are semantically prescribed at runtime as part of the recovery pattern: if the exception is an unchecked exception, no further constraints are required. If the exception is a checked exception, however, it must be declared by the method's exception attribute. As the exception attribute is given as a list of class types (in the class file) their types remain the same at runtime. Again, in parallel to Definition 4.4.4, three situations can appear at compile-time for each declared exception type. We interpret type safety in this case in the most conservative way, as we require the exception's compile-time type to be strictly less or equal to at least one of the class types, listed in the attribute.

In the previous discussion we specifically analyzed what it means to have type safety at an exception handler. However, in order to obtain type safety for an exception with respect to the entire exception table at compile-time, we must decide how statically to interpret an exception table recovery-pattern.

Discussion 4.4.5 (An Exception Verification Strategy). At runtime, a raised exception, checked or unchecked, must be handled by *the first handler in the linearly ordered exception table* which matches the exception at runtime [30, §2.16.2]. At compile-time, this translates into a linear, left to right search for a handle in a “catch situation” with respect to the raised exception's compile-time type. Whereas a “definite catch” situation, or a “no catch” situation at compile-time can be given a clear type safety interpretation as specified in Definition 4.4.4, potential catches are subject to further discussion.

If we decide only to establish type safety for the first handle which can be identified as a “definite catch” at compile-time, we obviously may risk to surpass the establishment of type safety in a “potential catch” which eventually could turn out to be the first matching handle in the exception table at runtime. On the other hand, if we require type safety to be established rigorously, not only for a handle of “definite catch” but for any handle in the exception table which can be identified as a “potential catch”, we may risk to reject type safe programs, because of type safety violations which may never come into play at runtime.

In our approach to type safety for exceptions, we choose the most conservative approach to type safety. Thus, we will require type safety to be established for *all possible catch patterns*, even if it means that a type safe program occasionally is rejected. Specifically, it means that we will pursue the following compile-time, type safety strategy in our exception formalizations.

- If the first handle to match the exception in the exception table is identified as a *definite catch*, it is sufficient to establish type safety for that handle, as discussed in (4.4.3) , in order to assure general type safety for the exception.
- If the first handle to match the exception in the exception table is identified as a *potential catch*, it is necessary, but *not sufficient*, to establish a type safety guarantee at that handle. In order to assure general type safety, we must require type safety to be established for the exception with respect to the remaining exception table.
- If no handle in the exception table matches the exception in a definite catch, no further type safety requirements are needed if the exception is declared as unchecked. If the exception is defined as checked, however, we take the most *conservative approach* to type safety for exception attributes, as already addressed in (4.4.3).

Before we proceed with the actual exception verification formalization, we must identify the group of checked exceptions and unchecked exceptions for our instruction subset. A grouping which depends on whether the official JVM specification prescribes compile-time recovery checks to be performed or not [30, § 11.2].

Discussion 4.4.6 (Checked and Unchecked, Subset Exceptions). In accordance with the Remark 4.3.18, our instruction subset can potentially raise the exceptions which are listed by their compile-time types in ES_{RunExc} , as well as any user-declared and user-thrown subclass instance of `Exception`. The compile-time types of the exceptions which the considered instruction subset may raise, can formally be grouped according to their recover semantics, *cf.*, the official JVM specification [30, § 11.2].

Checked Exceptions. We consider instances with compile-time type `Exception`, or any of its subclasses, which are *user-declared and user-raised*.¹¹

Unchecked Exceptions. Furthermore, we consider instances with compile-time type `RuntimeException` which the runtime-system may raise for our instruction subset.

$$ES_{\text{RunExc}} \stackrel{\text{def}}{=} \{ \text{NegativeArraySizeException}, \\ \text{ArrayStoreException}, \\ \text{ArrayIndexOutOfBoundsException}, \\ \text{ClassCastException}, \\ \text{NullPointerException} \}$$

We notice, that this group also includes the *user-declared and user-raised* exceptions which are statically given as subclasses of `RuntimeException`.

We supplement this with an important remark on the verification of unchecked exceptions.

¹¹A user-raised exception is thrown by the `athrow` instruction.

Remark 4.4.7 (Uncaught exception handling). The official argument for dividing exceptions into two categories is that the group of uncaught exceptions is an enormously big exception group, which it therefore makes no sense to “check” at compile time [30, § 11.2]. With the verification strategy in Discussion 4.4.5, we have a mean for simplifying the verification process, in that the exception table gives a simple, and limited number of exceptions to look for. This relies on the observation, that we only need to be concerned with those exceptions which may be handled, as type safety is to be ensured at a handling position.

Recall that our grouping, for our convenience, has omitted some smaller exception sets:

- The class `Error`, or any of its subclasses. We refer to Remark 4.4.22 for further details on the matter.
- System-raised¹² subclasses of the class `Exception`, *e.g.*, `EOFException`, or `MalformedURLException`.

In order to ease the readability of the formalizations, we introduce an abbreviated exception verification context.

Definition 4.4.8 (The Exception Verification Context).

$$(4.4.8a) \quad \Theta \in \text{ExcContext}_{\text{MS,ML}} = \text{CodeContext}_{\text{MS,ML}} \times \text{LocalType}_{\text{ML}} \times \text{Propagate}$$

$$(4.4.8b) \quad \text{PR} \in \text{Propagate} ::= \text{true} \mid \text{false}$$

where we omit the subscripts from the sort names, whenever the meaning is clear from the context.

We shall briefly comment on the composition of the `ExcContext` sort. Besides the `CodeContext`, we have specified a `LocalType` sort, which formalizes the expected local type at the handling location. According to the semantics of exceptions, we know that the expected operand stack only contains the raised exception. Thus there is no need to record the entire, expected frame type in the exception context. The third composition, `Propagate`, is introduced to formally mark whether a raised exception must be verified by the exception attribute or not. As we only verify those checked subclasses of `Exception` which are *user-raised*, *i.e.*, raised by `throw`, and as we only know whether it has been user-raised when we evaluate the `throw` rules in Definition 4.3.24, we need to propagate that information in case the exception is uncaught, through the proof-unfoldings. In fact, `PR` is only set to `true` by the rule in (4.3.24b).

Before we proceed, we list the formalities which our formalization must adhere to.

Notation 4.4.9 (Assumptions and notational conventions). We adopt the same assumptions and notational conventions as listed in the Definition 4.2.5, in the Definition 4.3.4, and in the Definition 4.3.9, without further modification.

Specifically we recall an important consequence of an earlier adopted verification assumption, which allows us to consider type safety at a handling location, without further formalities.

Proposition 4.4.10 (Handler type-safety is well-defined). A method frame type assignment `FTA`, is well defined in any of the method’s exception handling locations.

¹²A system-raised exception is thrown by the runtime-system or the virtual machine.

Rule	Catch	Safety constraint at PP''
(4.4.12a)	no	—
(4.4.12b)	no	—
(4.4.12c)	no	—
(4.4.12d)	no	—
(4.4.12e)	no	—
(4.4.14a)	definite	$CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP}$
(4.4.16a)	potential	$CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP}$

Figure 4.4: Type safety for exceptions raised at PP.

Proof. As the exception table is assumed to be well-formed prior to verification, we can assume that any of its exception handlers, say at location PP'' , represents a well-defined opcode position within the code, and as such is encompassed by the method FTA. \square

The described exception verification strategy, is formalized in Definition 4.4.12a through Definition 4.4.20. Since there are often more than one exception which can be thrown from a program point, we have designed the verifying inference system so that it verifies *a set of exceptions* rather than just one exception at a time. The five definitions, which cover five different verification situations: a “no catch”, a “definite catch”, and a “potential catch” situation at a handle, the verification of an uncaught exception, and finally, the specific verification of an uncaught, checked exception, which must be declared by the exception attribute.

In Figure 4.4, we have listed the inference rules which cover the first three verification situations, and the catch situation they correspond to. In the final column, we have listed the introduced type safety constraints as a function of those rules. As expected, we have that only if the current handle catches the raised exception in a “definite catch” or a “potential catch”, type safety has to be established at the handling location.

Definition 4.4.11 (Exception Verification Signature). An exception verification judgment signature is given as follows.

$$\boxed{\text{ExcContext} \vdash_{bv} \text{PPoint}, \text{Excs}, \text{ExcHandlers}}$$

where “ $\Theta \vdash PP, ES, EHS$ ” reads: the set of compile-time exceptions, ES , which may be raised at the program point PP , verifies on a list of the exception table handlers EHS , within the exception verification context Θ .

There are five inference rules which formalize verification of an exception set, raised at the same program point, which are given in a non-catch situation. The rules in (4.4.12a) and (4.4.12b), formalizes a situation where all exception handlers in the exception table has been verified, either because the exception table is empty, or because none of the exception handlers in the table has caught the currently verified exception in a definite catch. Thus, the current exception must have its propagation characteristics verified in a premise, performed by the inference system in Definition 4.4.19. Rule (4.4.12c) formalizes when all raised exceptions has been verified, and as such is given as an axiom. The rules in (4.4.12d) and (4.4.12e), finally formalize two non-catch situations

where the current handle is surpassed. Either because the handle-range is inappropriate, or because the handle cannot catch the exception.

Definition 4.4.12 (No Exception Catch). Take the judgment signature to be given as in Definition 4.4.11.

$$(4.4.12a) \quad \frac{CH \vdash_{bv} PR, CID_E, EA}{\Theta \vdash PP, CID_E \cdot ES, EHS}$$

$$(4.4.12a.i) \quad \text{where } \Theta = \langle \langle \langle -, CH \rangle, \langle -, EA, \epsilon \rangle, - \rangle, -, PR \rangle$$

$$(4.4.12b) \quad \frac{CH \vdash PR, CID_E, EA \quad \Theta \vdash PP, ES, ET}{\Theta \vdash PP, CID_E \cdot ES, \epsilon}$$

$$(4.4.12b.i) \quad \text{where } \Theta = \langle \langle \langle -, CH \rangle, \langle -, EA, ET \rangle, - \rangle, -, PR \rangle$$

$$(4.4.12b.ii) \quad ET = EH \cdot EHS$$

$$(4.4.12c) \quad \frac{}{\Theta \vdash PP, \epsilon, EHS}$$

$$(4.4.12d) \quad \frac{\Theta \vdash PP, CID_E \cdot ES, EHS}{\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS}$$

$$(4.4.12d.i) \quad \text{where } EH = \langle \langle PP_1, PP_2 \rangle, -, - \rangle$$

$$(4.4.12d.ii) \quad PP < PP_1 \vee PP \geq PP_2$$

$$(4.4.12e) \quad \frac{\Theta \vdash PP, CID_E \cdot ES, EHS}{\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS}$$

$$(4.4.12e.i) \quad \text{where } \Theta = \langle \langle \langle -, CH \rangle, -, - \rangle, -, - \rangle$$

$$(4.4.12e.ii) \quad EH = \langle \langle PP_1, PP_2 \rangle, -, CID'_E \rangle$$

$$(4.4.12e.iii) \quad PP_1 \leq PP < PP_2$$

$$(4.4.12e.iv) \quad CID_E \not\leq_{CH} CID'_E$$

$$(4.4.12e.v) \quad CID'_E \not\leq_{CH} CID_E$$

Remark 4.4.13. The rule in (4.4.12a) deserves special mentioning, as it assumes some additional knowledge. If we compare this rule with the one in (4.4.12b), we notice that by unfolding the verification proof for the first exception CID_E in the set, we assume to know whether the entire set $CID_E \cdot ES$ verifies or not. An assumption which relies on the observation in Section 4.3, that only the athrow-rule in (4.3.24b) can set the propagation variable PR , in which case *only one exception* is thrown. Thus, there is only one exception to verify by our rule, *i.e.*, ES is empty. Any other instruction which may raise the exception set, will have set PR to `false`, in which case there is nothing further to verify for either exception in the set, according to the axiom in (4.4.19a).

The verification of an exception which is definitely caught by some handler, is formalized by two premises: one which specifies type safety at the handling location, and one which recursively verifies the remaining, unverified exceptions, with the method's exception table. The type safety premise is given as a type assignment compatibility rule, which establishes that a method frame type assignment must be assignment compatible with the expected frame type at the handling location.

Definition 4.4.14 (Definite Exception Catch). Take the judgment signature to be given as in Definition 4.4.11.

$$(4.4.14a) \quad \frac{\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}} \quad \Theta \vdash \text{PP}, \text{ES}, \text{EHS}}{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}}$$

$$(4.4.14a.i) \quad \text{where} \quad \Theta = \langle \Omega, \text{LT}, - \rangle$$

$$(4.4.14a.ii) \quad \Omega = \langle \langle -, \text{CH} \rangle, \langle -, -, \text{ET} \rangle, \text{FTA} \rangle$$

$$(4.4.14a.iii) \quad \text{EH} = \langle \langle \text{PP}_1, \text{PP}_2 \rangle, \text{PP}'', \text{CID}'_E \rangle$$

$$(4.4.14a.iv) \quad \text{FT}_{\text{PP}''}^{\text{PP}} = \langle \text{CID}_E, \text{LT} \rangle$$

$$(4.4.14a.v) \quad \text{PP}_1 \leq \text{PP} < \text{PP}_2$$

$$(4.4.14a.vi) \quad \text{CID}_E \leq_{:\text{CH}} \text{CID}'_E$$

Remark 4.4.15 (Dynamic stack check). We notice that the type assignment compatibility check which is performed by the upper premise, could be reduced to a check of the local type assignment compatibility, since the stack is verified dynamically anyway.

The verification of an exception which is potentially caught by some handler, is formalized by two premises: one which, like the definite catch formalization, specifies type safety at the handling location, one which verifies the current exception with the remaining handlers in the exception table.

Definition 4.4.16 (Potential Exception Catch). Take the judgment signature to be given as in Definition 4.4.11.

$$(4.4.16a) \quad \frac{\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}} \quad \Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}}{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}}$$

$$(4.4.16a.i) \quad \text{where} \quad \Theta = \langle \Omega, \text{LT}, - \rangle$$

$$(4.4.16a.ii) \quad \Omega = \langle \langle -, \text{CH} \rangle, -, \text{FTA} \rangle$$

$$(4.4.16a.iii) \quad \text{EH} = \langle \langle \text{PP}_1, \text{PP}_2 \rangle, \text{PP}'', \text{CID}'_E \rangle$$

$$(4.4.16a.iv) \quad \text{FT}_{\text{PP}''}^{\text{PP}} = \langle \text{CID}_E, \text{LT} \rangle$$

$$(4.4.16a.v) \quad \text{PP}_1 \leq \text{PP} < \text{PP}_2$$

$$(4.4.16a.vi) \quad \text{CID}'_E <_{:\text{CH}} \text{CID}_E$$

Remark 4.4.17 (Dynamic stack check). Same comment on the assignment compatibility check as in Remark 4.4.15.

Remark 4.4.18. PP_2 formalizes the `end_pc` of the range in the method code array at which the exception handler EH is active. Since the `end_pc` is not included in this range, PP_2 is excluded from the try-range in the side conditions in our formalization of exception handlers [30, p.122].

We formalize the verification of uncaught exceptions by two rules: if the exception can propagate without being declared by the exception attribute (PR is `false`), or it can propagate if it is declared by the exception attribute (PR is `true`).

Definition 4.4.19 (Uncaught Exception Verification). A judgment which verifies an uncaught exception is structured by the following signature.

$$\boxed{\text{ClassHier} \vdash_{bv} \text{Propagate}, \text{ClassIdent}, \text{ExcAtt}}$$

where “ $CH \vdash_{bv} PR, CID_E, EA$ ” reads: an uncaught exception with the compile-time type CID_E , and propagation property PR , is verified within the class hierarchy CH with the exception attribute EA .

$$(4.4.19a) \quad \frac{}{CH \vdash_{bv} \text{false}, CID_E, EA}$$

$$(4.4.19b) \quad \frac{CH \vdash CID_E, EA}{CH \vdash_{bv} \text{true}, CID_E, EA}$$

Finally we formalize the verification of the exception attribute. As pointed out in (4.4.3), we require that the uncaught, propagating exception matches one of the declared exception in a “definite match”. In proof terminology, this means that the attribute verification rules must assure that a proof will fail when no “definite match” is found in the exception attribute.

Definition 4.4.20 (Exception Attribute Verification). A judgment which verifies a propagating, checked exception is structured by the following signature.

$$\boxed{\text{ClassHier} \vdash_{bv} \text{ClassIdent}, \text{ExcAtt}}$$

where “ $CH \vdash CID_E, EA$ ” reads: a propagating, checked exception of compile-time type CID_E , is checked with an exception attribute EA , within a class hierarchy CH .

$$(4.4.20a) \quad \frac{}{CH \vdash CID_E, CID'_E \cdot EA}$$

$$(4.4.20a.i) \quad \text{where} \quad CID_E \leq_{:CH} CID'_E$$

$$(4.4.20b) \quad \frac{CH \vdash CID_E, EA}{CH \vdash CID_E, CID'_E \cdot EA}$$

$$(4.4.20b.i) \quad \text{where} \quad CID_E \not\leq_{:CH} CID'_E$$

Remark 4.4.21. We notice that since the attribute list is finite, all proofs generated by the judgment set will be decidable.

Discussion 4.4.22 (Error verification). An exception of type `Error` (or a subtype hereof) is typically raised by the Java Virtual Machine as a result of an internal error [24]. As such, they can be expected to be raised from any program point of a method. A formalization should therefore add an `Error`-verifying premise to any instruction verifying rule. As `Error` exceptions are specified as unchecked exceptions, they can be verified by the same exception judgment in (4.4.11), which formally verifies `RuntimeException`. In our formalization, we have not considered to verify `Error` (and its subclasses), in order to keep the presentation simple. If added, however, we refer to Remark 4.4.7 for a way to manage unchecked exceptions.

4.5 The Example

Finally, we extend our canonical checksum example to the formalization system introduced in this chapter.

Figure 4.5 proposes a frame type assignment, FTA, for the `cksum()` method code listed in Example 3.4. In order to enhance readability, the table has been divided into six basic blocks¹³. The first column gives the relative program point positions of the Opcodes, as indicated by the method instruction sequence of column four. The two middle columns indicate the approximated frame types in terms of a frame type assignment stack and local variables for the method as the frame type assignment's local type name with the source-code variable names. Thus, a line reads as follows: at a program point (formalized as PP) is associated the frame type assignment (formalized as FTA), consisting of an approximated stack type (formally given by $FTA_{PP}.ST$) and a local type (formally given by $FTA_{PP}.LT$), on which the associated instruction (formalized as i) is defined.

Proposition 4.5.1 (The checksum method standard verifies). The checksum method `cksum()`, formalized by M^{ck} , bytecode verifies in the context Γ^{ck} (both specified in Figure 4.6) with the frame type assignment FTA^{ck} (defined in Figure 4.5). Formally speaking we have that

$$(4.5.1a) \quad \Gamma^{ck} \vdash_{bv} M^{ck}, FTA^{ck}$$

Proof. Assume Γ^{ck} , M^{ck} , and FTA^{ck} as in the Proposition. Then the verification rules of this chapter can be used to prove

$$(4.5.15) \quad \frac{\frac{}{CH^{ck} \vdash FTA_0^{ck} \sqsubseteq FT_0^{ck}} \text{ (4.1.28a)} \quad \frac{\boxed{\text{A.1.2}}}{\Omega^{ck} \vdash 0, C^{ck}, \emptyset \Rightarrow PPS^{ck}} \text{ (4.2.8b)}}{\Gamma^{ck} \vdash M^{ck}, FTA^{ck}} \text{ (4.2.7a)}$$

where

$$(4.5.15.i) \quad \Delta^{ck} = \langle \text{int}, MFR^{ck}, EA^{ck}, ET^{ck} \rangle$$

¹³A “basic block” is a code sequence where only the first instruction is a jump target (which includes the initial method invocation jump).

PP	FTA _{pp} .ST	FTA _{pp} .LT (this · cnum · x · y · z)	I
0	ε	Gcd11 · CrCardRd · ⊥ · ⊥ · ⊥	aload[1]
2	CrCardRd	Gcd11 · CrCardRd · ⊥ · ⊥ · ⊥	invokevirtual[1]
5	int	Gcd11 · CrCardRd · ⊥ · ⊥ · ⊥	istore[2]
7	ε	Gcd11 · CrCardRd · int · ⊥ · ⊥	goto[+8]
10	UnsetCrCard	Gcd11 · CrCardRd · ⊥ · ⊥ · ⊥	pop
11	ε	Gcd11 · CrCardRd · ⊥ · ⊥ · ⊥	new[2]
14	Abort	Gcd11 · CrCardRd · ⊥ · ⊥ · ⊥	athrow
15	ε	Gcd11 · CrCardRd · int · ⊥ · ⊥	ldc_w[3]
18	int	Gcd11 · CrCardRd · int · ⊥ · ⊥	istore[3]
20	ε	Gcd11 · CrCardRd · int · int · ⊥	iload[2]
22	int	Gcd11 · CrCardRd · int · int · ⊥	iload[3]
24	int · int	Gcd11 · CrCardRd · int · int · ⊥	isub
25	int	Gcd11 · CrCardRd · int · int · ⊥	istore[4]
27	ε	Gcd11 · CrCardRd · int · int · int	iload[4]
29	int	Gcd11 · CrCardRd · int · int · int	ifle[+10]
32	ε	Gcd11 · CrCardRd · int · int · int	iload[4]
34	int	Gcd11 · CrCardRd · int · int · int	istore[2]
36	ε	Gcd11 · CrCardRd · int · int · int	goto[-16]
39	ε	Gcd11 · CrCardRd · int · int · int	iload[4]
41	int	Gcd11 · CrCardRd · int · int · int	ifne[+6]
44	ε	Gcd11 · CrCardRd · int · int · int	iload[2]
46	int	Gcd11 · CrCardRd · int · int · int	ireturn
47	ε	Gcd11 · CrCardRd · int · int · int	iload[2]
49	int	Gcd11 · CrCardRd · int · int · int	istore[4]
51	ε	Gcd11 · CrCardRd · int · int · int	iload[3]
53	int	Gcd11 · CrCardRd · int · int · int	istore[2]
55	ε	Gcd11 · CrCardRd · int · int · int	iload[4]
57	int	Gcd11 · CrCardRd · int · int · int	istore[3]
59	ε	Gcd11 · CrCardRd · int · int · int	goto[-39]

Figure 4.5: A cksum() frame type assignment.

- (4.5.2) $\Gamma^{\text{ck}} = \langle \langle \text{CP}^{\text{ck}}, \text{Gcd11} \rangle, \text{CH}^{\text{ck}} \rangle$
- (4.5.3) $\text{CP}^{\text{ck}} = \{ 1 \mapsto \text{methodref}(\text{CrCardRd}, \text{methsig}(\text{getIt}, \epsilon), \text{int})$
 $2 \mapsto \text{Abort},$
 $3 \mapsto \text{int} \}$
- (4.5.4) $\text{CH}^{\text{ck}} = \{ \text{CrCardRd} \mapsto \text{Object}, \text{Gcd11} \mapsto \text{Object},$
 $\text{Abort} \mapsto \text{Exception}, \text{UnsetCrCard} \mapsto \text{Exception} \}$
- (4.5.5) $\text{M}^{\text{ck}} = \langle \text{MSIG}^{\text{ck}}, \text{int}, \text{EA}^{\text{ck}}, \text{CA}^{\text{ck}} \rangle$
- (4.5.6) $\text{EA}^{\text{ck}} = \langle \text{Abort} \rangle$
- (4.5.7) $\text{MSIG}^{\text{ck}} = \text{methsig}(\text{cksum}, \langle \text{CrCardRd} \rangle)$
- (4.5.8) $\text{CA}^{\text{ck}} = \langle \text{MFR}^{\text{ck}}, \text{C}^{\text{ck}}, \text{ET}^{\text{ck}} \rangle$
- (4.5.9) $\text{MFR}^{\text{ck}} = \langle 2, 5 \rangle$
- (4.5.10) $\text{C}^{\text{ck}} = \text{aload}[1] \text{ invokevirtual}[1] \text{ istore}[2] \text{ goto}[+8] \text{ pop}$
 $\text{ new}[2] \text{ athrow ldc_w}[3] \text{ istore}[3] \text{ iload}[2] \text{ iload}[3]$
 $\text{ isub istore}[4] \text{ iload}[4] \text{ ifle}[+10] \text{ iload}[4] \text{ istore}[2]$
 $\text{ goto}[-16] \text{ iload}[4] \text{ ifne}[+6] \text{ iload}[2] \text{ ireturn}$
 $\text{ iload}[2] \text{ istore}[4] \text{ iload}[3] \text{ istore}[2] \text{ iload}[4]$
 $\text{ istore}[3] \text{ goto}[-39]$
- (4.5.11) $\text{ET}^{\text{ck}} = \langle \langle \langle 0, 7 \rangle, 10, \text{UnsetCrCard} \rangle \rangle$
- (4.5.12) $\text{FT}_0^{\text{ck}} = \langle \epsilon, \langle \text{Gcd11}, \text{CrCardRd}, \perp, \perp, \perp \rangle \rangle$
- (4.5.13) $\text{PPS}^{\text{ck}} = \{ 0, 2, 5, 7, 10, 11, 14, 15, 18, 20, 22, 24, 25, 27,$
 $29, 32, 34, 36, 41, 44, 46, 47, 49, 51, 53, 55, 57, 59 \}$
- (4.5.14)

Figure 4.6: Context for `cksum()` verification.

$$(4.5.15.ii) \quad \Omega^{\text{ck}} = \langle \Gamma^{\text{ck}}, \Delta^{\text{ck}}, \text{FTA}^{\text{ck}} \rangle$$

and where A.1.2 denotes the full proof tree given in Appendix A.1.

□

Chapter 5

Lightweight Verification Formalization

In this chapter, we develop the idea of using type certificates to check the type safety of method bytecode, originally proposed by the author [46, 47]. This presentation extends a preliminary formalization by the author and K.H. Rose [48] in the following ways. First of all, the notion of “lightweight type safety” has been argued and formalized as a property of the Java type safety lattice. Second, the actual lightweight verification algorithm has been systematically formalized for the JVM subset. The formal proof which finally links lightweight bytecode verification and bytecode type safety together, however, is postponed to Chapter 6, when we formally introduce certificates as “type safety guarantees”.

In Section 5.1 we present and formally analyse the idea of lightweight bytecode type safety and verification. In Section 5.2 we discuss our verification assumptions and present the semantics for method bytecode lightweight verification. In Section 5.3 we semantically continue our lightweight type safety formalizations with a formalization for our instruction subset, and in Section 5.4, we formalize “lightweight type safety” for exceptions. Finally, in Section 5.5, we present the complete lightweight verification proof-unfolding for our canonical checksum method.

5.1 Analysis and Formalization Strategy

“Lightweight bytecode verification” assumes that it is possible, from a generally small set of frame type annotations (a lightweight certificate), using the “built-in” type system of Java bytecode to type check whether a method is type safe; and that in *one, straight code pass*. In other words: if a method bytecode can be verified by an arbitrary dataflow algorithm, we postulate it is possible to verify type safety in a *fixed constraint solving order*, based on *the existence of* a set of frame type annotations which is *not proportional* to the code size. We notice that

- frame type constraints are only build up in a non-trivial way at jump targets during standard bytecode verification, and
- the order in which these constraints are solved is insignificant.

The first observation comes from the fact that the standard verifier is specified as a dataflow algorithm, which means that all jumps target an entrance to a basic block [30, 35]. The observation actually suggests that it is sufficient for type safety, to constraint solve the frame types at jump

targets. The second observation is due to Cousot and Cousot’s work on chaotic equations [11] as pointed out in the beginning of Section 4.2, something which is also reflected by our frame type type-safety model as stated in Lemma 4.1.38. The observation suggests that choosing a specific, say straight code pass, does not circumvent method bytecode type safety guarantees.

The first observation may in fact be formalized as follows.

Observation 5.1.1. Let FTA be a solution to the frame type constraint set given for a JVM method with the code component C . Let PP_1 be the first program point of an arbitrary basic block given by the code segment $C|_{\{PP_1, \dots, PP_k\}}$. We notice:

all frame types which can be expected from FTA (PP) within that basic block, is a solution to the constraint set on $C|_{\{PP_1, \dots, PP_k\}}$.

As a consequence we have that from the set of all labels¹ (and 0), given by $\{PP_0, \dots, PP_k\}$, which exists for some (verifiable) method code, and a solution FTA to the method’s frame type constraint set, we can actually construct another solution FTA’, from the frame type set containing $\{FTA(pp_0), \dots, FTA(pp_k)\}$ in one, linear pass through the method code.

Definition 5.1.2 (Base, Frametype Constraint Solution Set). Let a method have the method code C , and let $\{PP_0, \dots, PP_k\} \in \mathbf{P}(PPS_C)$. A frame type set given by $\{FT_{PP_0}, \dots, FT_{PP_k}\}$ from which a solution to the method’s frame type constraint set can be constructed, is called a *base, frame type constraint solution set* for that method.

In the following, we may refer to a “base frame type constraint solution set” simply as a “base solution set”, whenever the meaning is clear from the context.

Proposition 5.1.3 (The existence of a base solution set). If a method is bytecode verifiable, there exists a base, frame type constraint solution set, such that a solution to the method’s frame type constraint set can be iterated in one, linear pass through the code.

Proof. Assume that a method is verifiable. Then there exists a solution FTA to its frame type constraint set. According to Observation 5.1.1, we can construct a solution FTA’ from $FTA|_{\{PP_1, \dots, PP_k\}}$, where $\{PP_1, \dots, PP_k\}$ is the set of all labels in the method (and 0), in a stepwise manner, starting from program point 0. □

Remark 5.1.4. We notice that the idea of using a base solution set to verify a method, has been implemented in Sun’s KVM virtual machine [59].

In our approach, we work from the initial hypothesis that a “solution frame typing” is available together with the method code prior to bytecode verification. However, as also pointed out by Leroy [25], the effectiveness of a verification strategy which requires the presence of a complete base solution set, is not always as effective as one would like in practice. (The KVM verifier is estimated to be approximately 10 times the possible size to fit on a general smart card [20].) In this section, however, we intend to go further in reducing the solution set of needed frame type annotations (the certificate) in order to (re)assure bytecode type safety (through type checking). In order to suggest a strategy, we shall study an example.

¹In assembler technology, a label is a marker for a jump target (including exception handlers).

Example 5.1.5 (A jump situation). Consider the following sequence of expected frame types of dimensions $\langle 2, 3 \rangle$, which ends by a backward jump at $pp+5$, and let the class identifiers CID_1 , and CID_2 be given with respect to some class hierarchy CH .

PP	FTA(PP).ST	FTA(PP).LT	I
⋮			
PP	$x \cdot \text{int}$	$CID \cdot CID_1 \cdot CID_2$	pop
PP + 1	x	$CID \cdot CID_1 \cdot CID_2$	aload[1]
PP + 3	$x \cdot CID$	$CID \cdot CID_1 \cdot CID_2$	astore[2]
PP + 5	x	$CID \cdot CID_1 \cdot CID_2$	goto[-4]
⋮			

1. if $CID_2 \leq_{CH} CID_1$, the given frame typing is a current solution to the constraint in $PP+1$, as it adheres to the type safety constraints of (4.3.22a).
2. if $CID_2 \not\leq_{CH} CID_1$, however, the given frame typing is *not* a solution to the constraint in $PP + 1$, as it violates the type safety constraint $CH \vdash FTA_{pp''} \sqsubseteq FT''$ (i.e., the premise of (4.3.22a)).

In the latter case, an altered solution to the constraint imposed by the goto instruction must be found by “lowering” the current solution at $PP + 1$, in order to avoid the kind of type confusion situations as described in Example 4.1.2. Formally, a solution must include a frame type FT_{PP+1} such that $FT_{PP+1} \sqsubseteq FT_{PP+1}^{PP}$ and $FT_{PP+1} \sqsubseteq FT_{PP+1}^{PP+5}$. Since the ordered frame type set constitute a lattice, we know that at least one such frame type exists, and is given by the “meet” $FT_{PP+1}^{PP} \sqcap FT_{PP+1}^{PP+5}$.

PP	FTA(PP).ST	FTA(PP).LT	I
⋮			
PP	$x \cdot \text{int}$	$CID \cdot CID_1 \cdot (CID_1 \sqcap CID_2)$	pop
PP + 1	x	$CID \cdot CID_1 \cdot CID_1$	aload[1]
PP + 3	$x \cdot CID$	$CID \cdot CID_1 \cdot CID_1$	astore[2]
PP + 5	x	$CID \cdot CID_1 \cdot CID_1$	goto[-4]
⋮			

Finally we notice, that all of the above considerations in the example would have hold for a backward jump situation as well.

In Figure 5.1 we have listed the instructions and associated verification rules which deal with the establishment of type safety at a jump target during standard bytecode verification. None of which distinguishes between jump directions. The distinction between forward and backward jump targets becomes necessary, as the constructed frame type constraints is not met, hence cannot be solved in the same manner, when we traverse the method code in a specific direction. We have that

- A backward jump target is special in that it is reached *before* the actual jump appears during the stepwise method code scan, and thus the need for a constraint to be solved.

Rule	Jump occasion
(4.3.20a)	ifne[n], ifle[n], or ifnull[n]
(4.3.22a)	goto[n]
(4.4.14a)	Exception handling (definite catch)
(4.4.16a)	Exception handling (potential catch)

Rule	Exception occasion
(4.3.11a)	iaload, aaload, iastore, aastore, newarray_int, anewarray[n], arraylength
(4.3.13a)	checkcast[n]
(4.3.16a)	putfield[n], getfield[n]
(4.3.17a)	invokevirtual[n]
(4.3.24a), (4.3.24b), (4.3.24c)	athrow

Figure 5.1: Type safety establishment at jump targets.

- A forward jump target is special in that it is only reached *after* the jump has been detected during the stepwise method code scan, thus the need for a constraint to be resolved is known in advance.

In Figure 5.2, we have formulated these considerations in a description of how we initially imagine a lightweight verification process to perform. The program points and frame types which we imagine as externally provided, form what we call a lightweight certificate. Based on the presented discussions and observations in this section, we make the following important key-assumption, on which we base our formalizations.

Definition 5.1.6 (The Lightweight Assumption). We assume that it is possible to construct a solution to a verifiable method’s constraint set from an appropriate certificate in one, straight code pass.

Based on Figure 5.2, we give the following sort-algebraic specification of a certificate.

Definition 5.1.7 (A Lightweight Certificate). Let a method have the code component C and dimensions given by MS and ML . A “lightweight certificate” for the method is formally specified by

$$(5.1.7a) \quad CE \in \mathbf{Cert}_{C,MS,ML} = \mathbf{FrameTypeCert}_{C,MS,ML} \times \mathbf{Labels}_C$$

$$(5.1.7b) \quad LS \in \mathbf{Labels}_C = \mathbf{P}(\mathbf{Label}_C)$$

$$(5.1.7c) \quad \mathbf{FTC} \in \mathbf{FrameTypeCert}_{C,MS,ML} = \mathbf{PPoints}_C \rightarrow (\mathbf{FrameType}_{MS,ML})_{\perp}^{\top}$$

$$(5.1.7d) \quad \mathbf{Label}_C = \mathbf{PPoint}_C$$

Notation 5.1.8. The deduction of an “appropriate” lightweight certificate from a verifiable method is called “certification”, and is the subject of Chapter 6.

Backward Jump Targets cannot be predicted in advance during the stepwise lightweight verification procedure. Thus, there is no way for the lightweight procedure to record the associated frame types for a later constraint check, unless these program points are either *externally provided* (or provided by a second code scan).

Forward Jump Targets are detected in advance when a forward reference appears in the code. Thus, it is straightforward for the stepwise lightweight verification procedure to record the associated frame types for a later constraint check.

“Inappropriate” Program Points are typically those program points which follow non-fallthrough instructions and are not forward jump targets, *i.e.*, where no frame type can be constructed from the previous program points during a stepwise lightweight verification procedure. Inappropriate program points are generally speaking those program points to which it is not possible to construct a frame type which is part of a solution to the method’s constraint set. Thus, we suggest that the “appropriate” frame types for these program points are *externally provided*.

Figure 5.2: A “pre-certificate” description.

Notation 5.1.9 (The certificate components). In the following, we will refer to the set of registered backward jump target program points LS , as the “backward labels” or simply “the label set”. The frame type map FTC of “necessary” backward frame types in the certificate, is referred to as the “certificate frame types”

The two dynamic structures which we imagine to accumulate the current constraints from backward and forward jumps during a stepwise code scan, will be referred to as “pending” and “saved”. The purpose with “pending” is to accumulate the frame types which are imposed as constraints at forward jump targets. The purpose with “saved”, however, is to accumulate the frame types which are known at the backward jump targets when these are reached during the stepwise scan. In order to ensure that the suite of frame types which we step through by the linear code pass constitutes a solution to the method’s constraint set, we must require two things, as discussed in Example 5.1.5:

- any “pending” frame type must be greater (or equal) to the supposed frame type solution in the associated forward jump target, and
- any “saved” frame type, which by the lightweight assumption described in Definition 5.1.6, is supposed to be part of a solution to the method’s constraint set, must be smaller than the frame type constraint which is imposed by the associated backward jump.

Based on these considerations we give the following sort algebraic specification of the “pending” and “saved” structures.

Definition 5.1.10 (The Delayed Frame Type Constraint Sort). Let a method have the code component C and dimensions given by MS and ML . A “delayed constraint set” for the method, is

Base solution frame-types	Certificate	Safety handling
at backward labels	backward labels (LS)	“saved” (S)
at forward labels		“pending” (P)
at “inappropriate” program points	frame types (FTC)	

Figure 5.3: Altering the verification strategy.

formally specified by

$$(5.1.10a) \quad DC \in \text{DelayConstr}_{c,MS,ML} = \text{Pending}_{c,MS,ML} \times \text{Saved}_{c,MS,ML}$$

$$(5.1.10b) \quad P \in \text{Pending}_{c,MS,ML} = \text{FrameTypeMap}_{c,MS,ML}$$

$$(5.1.10c) \quad S \in \text{Saved}_{c,MS,ML} = \text{FrameTypeMap}_{c,MS,ML}$$

Remark 5.1.11. In Figure 5.3 we have listed how the idea of verification from a base solution set has been altered by the described lightweight verification components.

Based on Example 5.1.5, and the discussions on pending and saved, we explain what we understand by a current frame type in a program point.

Definition 5.1.12 (The Current Frame Type). A current frame type in a program point is given as the result of merging all of the currently available frame type constraints during lightweight verification.

As the relative jump target position plays an important role in lightweight verification, we specify its formal signification.

Definition 5.1.13 (Jump Directions). Let a *jump instruction* be given at a program point PP in a method, with a jump target PP'' .

A Backward Jump is defined when the jump target is given by $PP'' \leq PP$, *i.e.*, the target address is *less or equal*² to the address of the jump instruction.

A Forward Jump is defined when the jump target is given by $PP'' > PP$, *i.e.*, the target address is *strictly greater than* the address of the jump instruction.

Discussion 5.1.14 (Delayed type safety handling). Recall that type safety constraints primarily are build up at jump targets. First, let us consider how such constraints are handled by the standard bytecode verifier in Chapter 4. In Figure 5.1, we have made an exhaustive listing of those rules which, within our instruction subset, may require that type safety is handled at a jump target. (The “may” refer to the fact that not all exceptions which are raised, will be handled, and thus result in a control transfer to another location in the code.) A type safety verification which in each case is performed by the premise:

²A jump to the same location is mainly used for “busy waiting” in concurrent programming, typically in order to invite another thread to take over an evaluation. Since Java does not support this kind of behavior, the case has no practical relevance [30, §17].

$$(5.1.14a) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$$

where PP'' is the jump target, and PP is where the control transfer originates.

Since the order in which the frame type constraints are solved is insignificant with respect to type safety, we can exploit this in lightweight verification by separating the type safety constraint management for backward and forward jumps, and solving them in the order in which they appear during a stepwise code pass, beginning at program point 0. Specifically we suggest how “pending”, formalized by P , manages forward jump constraints, and “saved”, formalized by S , manages backward jump constraints. We illustrate the lightweight safety-constraint management idea in the following example. In order to keep the presentation simple, we postpone the explanation of the role of a lightweight certificate to Discussion 5.1.19. For the time being, we simply assume that the backward jump targets are known.

Example 5.1.15 (A lightweight procedure). We consider a code jump situation with just one forward jump and one backward jump, both to the instruction at PP .

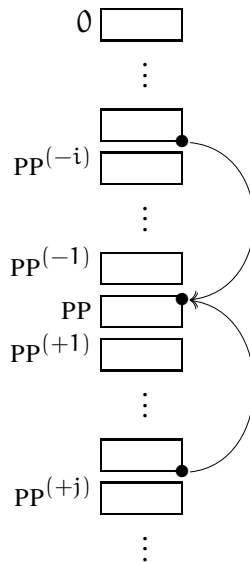


Figure 5.4: A code jump situation.

Figure 5.4 shows a method code component of at least two instructions.³ We assume an instruction at program point PP , a previous instruction at a program point $\text{PP}^{(-1)}$, and a successor instruction at a program point $\text{PP}^{(+1)}$. Furthermore, a jump instruction at $\text{PP}^{(-i)}$ targeting PP in a forward jump, *i.e.*, from $\text{PP}^{(-i)} < \text{PP}$, and a jump instruction at $\text{PP}^{(+j)}$ targeting PP in a backward jump, *i.e.*, from $\text{PP} \leq \text{PP}^{(+j)}$.

A lightweight verifier is thought to progressively verify the type safety of one instruction at a time, beginning with the instruction at program point 0.⁴ Let us chronologically investigate how such a stepwise procedure may progress in the situation of Figure 5.4 where all jumps are shown.

³There must be at least two instructions in a method to foster a jump.

⁴There is at least one instruction in any legal method.

- From the instruction at program point 0 to the instruction at $PP^{(-i)}$, there will not be a branching or confluence in the data flow, as there are no jump instructions or targeting program points until $PP^{(-i)}$. Thus, the current frame type at every program point, is given as a non-trivial,⁵ expected frame type constraint, which is uniquely given by the static semantics of the associated instruction.
- At $PP^{(-i)}$, the lightweight procedure detects a jump instruction, which is forward directed towards PP . The jump frame type, *i.e.*, $FT_{PP}^{PP^{(-i)}}$, which is the result of applying the jump instruction semantics to the current frame type at $PP^{(-i)}$, is hence *accumulated as the pending frame type of $P(PP)$* .
- From the instruction $PP^{(-i)}$ to the instruction at PP , there is no branching in the data flow, as there are no other jump instructions or targeting program points until PP . Thus, the current frame type at every program point, is given as a non-trivial, expected frame type constraint, which is uniquely given by the static semantics of the associated instruction. (Notice, that the jump instruction at $PP^{(-i)}$ cannot be a non-fall through instruction,⁶ as there are no jumps to the following instruction, and dead code is not allowed. Thus, the expected frame type for the following instruction will be non-trivial.)
- At PP , the lightweight verifier knows of two frame type constraints, the expected frame type $FT_{PP}^{PP^{(-1)}}$, imposed from the previous instruction at $PP^{(-1)}$, and the delayed, pending frame type $P(PP)$, imposed by the instruction at $PP^{(-i)}$. Thus, the current frame type at PP becomes $P(PP) \sqcap FT_{PP}$ (as already argued in Example 5.1.5). If the method is verifiable, we have according to (5.1.14a), that there exists a solution FTA such that

$$(5.1.15a) \quad CH \vdash FTA(PP) \sqsubseteq P(PP)_{PP^{(-i)}}$$

$$(5.1.15b) \quad CH \vdash FTA(PP) \sqsubseteq FT_{PP}^{PP^{(-1)}}$$

and therefore

$$(5.1.15c) \quad CH \vdash FTA(PP) \sqsubseteq P(PP)_{PP^{(-i)}} \sqcap FT_{PP}^{PP^{(-1)}}, \quad PP > PP^{(-i)}$$

where we have added the subscript $PP^{(-i)}$ in the formulas to indicate the origin of the pending forward jump.

- As PP is a backward target, the current type constraint at PP must be saved for any later (delayed) constraint checks with any backward imposed frame types. Thus, $P(PP) \sqcap FT_{PP}^{PP^{(-1)}}$ must be *registered as the saved frame type $S(PP)$* . According to the lightweight assumption we specifically have, that the saved, current type constraint at PP is *part of a solution to the method's constraint set at PP* .
- From the instruction at PP to the instruction at $PP^{(+j)}$, there is again no branching or confluence in the data flow, as there are no other jump instructions or targeting program points

⁵A non-trivial expected frame type is one which is not \top .

⁶All instructions in our instruction subset expects at least that the stack type length is semantically given. If we had included nop, however, this would not have been the case.

until $PP^{(+j)}$. Thus, the current frame type at every program point, is given as a non-trivial, expected frame type constraint, in parallel with the considerations above.

- At $PP^{(+j)}$, the lightweight procedure detects a jump instruction which is backward directed towards PP . The jump frame type, $FT_{PP}^{PP^{(+j)}}$, which is the result of applying the jump instruction semantics to the current frame type at $PP^{(+j)}$, is thus imposed at PP . According to the lightweight assumption, we have that if the method is verifiable, $S(PP)$ is a solution to the method's constraint set at PP , and thus, according to (5.1.14a), we have that

$$(5.1.15d) \quad CH \vdash S(PP) \sqsubseteq FT_{PP}^{PP^{(+j)}}, \quad PP < PP^{(+j)}$$

- Finally, from the instruction at $PP^{(+j)}$ to the last instruction of the method, there is again no branching or confluence in the data flow, as there are no other jump instructions or targeting program points. Thus, the current frame type at every program point, is given as a non-trivial, expected frame type constraint, in parallel with the considerations above.

Let us assume the same program point notation as in Example 5.1.15. As we generalize the type safety requirements of (5.1.15c) and (5.1.15d) to the complete set of forwards jumps and backward jumps to a program point PP over a method code C , we obtain the following propositions.

Proposition 5.1.16. For any PP , which is a forward jump target in a given, verifiable method with an arbitrary solution FTA , we have that the collection of all forward jumps from each of $PP^{(-1)}, \dots, PP^{(-k)}$ to the given target PP will result in a collection of k “pending” frame types $P(PP)_{PP^{(-i)}}$ from $PP^{(-i)}$ for $i \in \{1, \dots, k\}$, which satisfies the following constraint

$$CH \vdash FTA(PP) \sqsubseteq \left(\prod_{i=1}^k P(PP)_{PP^{(-i)}} \sqcap FT_{PP}^{PP^{(-1)}} \right)$$

Proof. Follows from Lemma 4.1.38 and the definition of \sqcap on a complete lattice. \square

Proposition 5.1.17. For any PP which is a backward jump target in a given, verifiable method, we have that if the “saved” frame type in PP is part of a solution to the method's constraint set, the collection of all backward jumps from each of $PP^{(+1)}, \dots, PP^{(+l)}$ to PP , will satisfy the following constraint

$$CH \vdash S(PP) \sqsubseteq \prod_{j=1}^l FT_{PP}^{PP^{(+j)}}$$

Proof. Follows from Lemma 4.1.38 and the definition of \sqcap on a complete lattice. \square

Remark 5.1.18 (Constraint invariants). We notice, that each of the frame types $P(PP)_{PP^{(-i)}}$ for $i \in \{1, \dots, k\}$ in Proposition 5.1.16, are known when we lightweight evaluate the instruction at PP , as well as the expected frame type $FT_{PP}^{PP^{(-1)}}$, which is given by the semantics of the preceding instruction. Thus, an eventually saved frame type $S(PP)$ is also known when verifying the instruction at PP .

The Proposition 5.1.16 and Proposition 5.1.17 suggest that with several jumps to the same location, the order in which the “pending” or “saved” constraints are introduced is insignificant.

Discussion 5.1.19 (A formalization strategy). In the Example 5.1.15, we assumed that the position of the backward target was known in advance. As earlier mentioned, the set of backward labels in a method is assumed to be provided by the label certificate component, formalized by LS.

Furthermore, the example indicated how the current frame type in every program point PP, can be found from local frame type information to be a non-trivial frame type. Imagine, however, a jump situation where the instruction at PP which follows a non-fall through instruction, is only targeted by a backward jump. As the lightweight technique which we sketched in Example 5.1.15 only depends on local frame type information, the current frame type at PP becomes the same as the expected frame type, *i.e.*, \top , which is unacceptable regardless of the following instruction (unless it is a simple return and, if we had included it, the `nop` instruction). A situation like this is where the frame type certificate component, formalized by FTC, comes in helpfully. In Section 6.1 we shall discuss the formal design issues for an appropriate certificate. For the time being, however, we simply list the principles, after which a certificate is deduced. Let C be a code component of some verifiable method, and FTA a solution to the method’s constraint set. Let the assumptions be as in Proposition 5.1.16.

- If the instruction at PP is a jump instruction and PP'' is the jump target, we have that if $PP'' \leq PP$ then $PP'' \in LS$ by our certification strategy.
- Whenever $FTA(PP) \sqsubseteq \left(\prod_{i=1}^k P(PP)_{PP(-i)} \sqcap FT_{PP}^{PP(-1)} \right)$, we have that FTC is updated with FTA in PP.

In other words, if the constraint which naively can be found by the lightweight verifier, that is $\prod_{i=1}^k P(PP)_{PP(-i)} \sqcap FT_{PP}^{PP(-1)}$, differs from the solution $FTA(PP)$ from which the certificate was generated, we externally impose the solution’s frame type in terms of FTC at PP. If we let FTC be a total map initialized as

$$(5.1.19a) \quad FTC_0 = \{PP \mapsto \top \mid PP \in PPS_c\}$$

we have that the following constraint is satisfied for all program points

$$(5.1.19b) \quad CH \vdash FTA(PP) \sqsubseteq \left(FTC(PP) \sqcap \prod_{i=1}^k P(PP)_{PP(-i)} \sqcap FT_{PP}^{PP(-1)} \right)$$

Specifically we have, that this certification strategy captures the problem of the stepwise procession of the lightweight verifier beyond non-fall through instructions.

We notice, that it is possible to produce a “false” frame type certificate for a given method which contains dead code, which allows the method to pass by the lightweight verifier. However, as discussed in Chapter 3, dead code is not harmful on its own, so in order to simplify the approach, we have decided not to consider the problems related to rejecting method’s which contains dead code. (In fact, if the size of the method becomes a problem, the system in general will reject the code at an earlier stage.)

Non-target The only main constraint imposed at a non-target PP is the expected frame type $FT_{PP}^{(-1)}$. The current frame type is adjusted by the frame type certificate in PP, *i.e.*, $FTC(PP) \sqcap FT_{PP}^{PP^{(-1)}}$.

Forward target There are two main constraints imposed at a forward target PP: a pending constraint as well as an expected frame type in that program point. The current frame type is adjusted by the frame type certificate in PP, *i.e.*, $FTC(PP) \sqcap P(PP) \sqcap FT_{PP}^{PP^{(-1)}}$.

Backward target The current frame type, imposed at a backward target PP is saved as $S(PP) = FTC(PP) \sqcap FT_{PP}^{PP^{(-1)}}$, until the lightweight verifier can check the confluence constraints as it reaches the jump source(s).

Forward and backward target combines the two previous cases. Thus, the current frame type becomes given by the most restrictive frame type of the two cases, and saved as $S(PP) = FTC(PP) \sqcap P(PP) \sqcap FT_{PP}^{PP^{(-1)}}$.

Figure 5.5: Confluence analysis.

Let us make some remarks on the S and P frame type element “garbage collection” during a lightweight verification procedure. Whenever an instruction in some method code C, at some forward jump PP' has been lightweight verified, it is of no further interest for the verifier, as it passes the code in a stepwise fashion. Thus, all pending frame types, associated to program points PP, where $PP \leq PP'$ for all $PP \in PPS_C$ can be eliminated without affecting the lightweight verification. For instructions at program points which are marked as backward labels, however, we do not know, during a single code scan, how many times a backward label will be targeted. Thus, we cannot allow any of the “saved” frame types to be removed as long as the code is being lightweight verified. If lightweight verification were to take place in two code passes instead of one, however, we could add a reference count number to each backward target. A number which could tell us, when a program point will not be targeted any more during lightweight verification, and hence could safely be removed. In order to keep our approach simple, however, we will not consider reference counting in this formalization. Thus, we cannot remove any saved frame type elements during the presented lightweight verification formalization. With these requirements in mind, we propose P and S to be initialized by the following maps.

$$(5.1.19c) \quad P_0 = \{PP \mapsto \top \mid PP \in PPS_C\}$$

$$(5.1.19d) \quad S_0 = \{PP \mapsto \perp \mid PP \in PPS_C\}$$

Notice that with those initialization maps, we suggest that the delayed constraints in (5.1.17) and (5.1.16) are trivially satisfied by all program points in a method code to begin with.

In Definition 2.3.8 we defined what is a function “meet” or “join” with a map over a singleton domain. With the initialization maps P_0 and S_0 , we can now formally define what it means to *extend or update* P and S with a frame type FT in a single program point PP: $P \sqcap \{PP \mapsto FT\}$, and

Forward jump instruction at program point PP , targeting $PP'' > PP$. We have that the resulting frame type $FT_{PP''}^{PP}$ from applying the static instruction semantics on the current frame type at PP , is extending the pending map $P' = P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$ at the target.

Backward jump instruction at program point PP , targeting $PP'' \leq PP$. We have that the resulting frame type $FT_{PP''}^{PP}$ from applying the static instruction semantics on the current frame type at PP , is checked with the saved constraint at PP'' . Thus, $S(PP'') \sqsubseteq FT_{PP''}^{PP}$.

Figure 5.6: Branching analysis.

$S \sqcup \{PP \mapsto FT\}$. We can also define what it means to *reduce or “garbage collect”* P and S in a single program point given by PP : $P \sqcup \{PP \mapsto \top\}$, and $S \sqcap \{PP \mapsto \perp\}$.

Finally, we need to structure and formalize those jump events which changes the data flow and hence the current frame type from the expected.

In Figure 5.5 we have constraint-analysed the four different data flow confluence situations which *every program point* of a method belongs to. Furthermore, we have constraint-analysed the two kinds of data flow branching situations in Figure 5.6, which *the jump instructions* of a method will result in.

In order to ensure that we can update S and P with a frame type map element we must assure that

Corollary 5.1.20. *Given a class hierarchy CH , a method with a code component C , dimensions $\langle MS, ML \rangle$, and two total, delayed frame type maps P , and S , we have that for a map $\{PP \mapsto FT\} \in \text{FrameTypeMap}_{C,MS,ML}$, over some singleton domain $\{PP \mid PP \in PPS_C\}$ that*

- $P \sqcap \{PP \mapsto FT\}, P \sqcup \{PP \mapsto \top\}$ are well-defined in $\text{FrameTypeMap}_{C,MS,ML}$
- $S \sqcup \{PP \mapsto FT\}, S \sqcap \{PP \mapsto \perp\}$ are well-defined in $\text{FrameTypeMap}_{C,MS,ML}$

Proof. Follows from the definition in (2.3.8) of a function “meet” and “join” with a map over a singleton domain, and the fact that $\langle \sqsubseteq_{MS,ML}, \text{FrameType}_{MS,ML} \rangle$ defines a lattice (which ensure the existence of “meet” and “join” over the ordered frame type set.) \square

We summarize the modifications which formally may be performed on P and S , as explained in Discussion 5.1.19.

Definition 5.1.21 (Delayed Constraint Map Modifications). With the initialization maps of the lightweight delayed frame type constraint maps given in (5.1.19d) and (5.1.19c), we have that

- $P \sqcap \{PP \mapsto FT\}$ and $S \sqcup \{PP \mapsto FT\}$ formally indicate the extension of P and S at the program point PP with a frame type FT .
- $P \sqcup \{PP \mapsto \top\}$ and $S \sqcap \{PP \mapsto \perp\}$ formally indicate the reduction of P and S at the program point PP .

5.2 Bytecode Verification

Standard bytecode verification was formalized in Chapter 4 to serve as a formal basis for a lightweight verification. In order to compare the two verification approaches, we will apply the same formal methodology of natural semantics [22]. Thus, lightweight verification, informally resumed by the assumption in Definition 5.1.6, is formalized as a big step, operational semantics. A formalism which is intended at the construction of a (logical) proof for a method code component and a lightweight certificate, within a *finite number of exhaustive judgment unfoldings*, provided that the method code lightweight verifies with the given certificate. (The proof which shows the lightweight verification formalization indeed provides the same type safety guarantees as standard lightweight verification, is postponed to Chapter 6.)

The actual lightweight verification system is specified by Definitions 5.2.5 through 5.4.7. The inference rules are obtained from the standard verification system specified by Definitions 4.2.7 through 4.4.11, basically by the exchange of the the frame type assignment component FTA with the frame type certificate component FTC, and by an extension of the standard verification judgment syntax and semantics with the specific lightweight components: a lightweight certificate , a pending map, and a saved map, as specified in the discussion on a formalization strategy in (5.1.19).

Before we proceed, let us briefly list the assumptions and notational conventions which we assume in our formalization of lightweight verification.

Notation 5.2.1 (Assumptions and notational conventions). The lightweight formalization must adhere to the following requirements.

- Same requirements as for standard verification in Notation 4.2.5.
- Same conventions as listed in Notation 4.3.4.
- *By syntactic abuse of notation*, we allow $S(\text{PP})$, $P(\text{PP})$, as well as $\text{FTC}(\text{PP})$ for a given $\text{PP} \in \text{PPoint}$ in lightweight judgments and inference rules. We notice that a correct but more cumbersome approach, would be either to introduce function application syntactically, or to deal with it in the inference-rule side conditions.
- *By syntactic abuse of notation*, we allow “meet” types of the kind $\text{FT} \sqcap \text{FT}'$ in side conditions, in order to simplify the formalization presentations. We notice that a correct but more cumbersome approach, would be to specify $\text{CH} \vdash \text{FT} \sqcap \text{FT}'$ as a premise.
- A code segment denotes a consecutive sequence of instructions of a method’s code component. In our formalizations, a code segment always include the last method instruction.
- We assume that FTC has been initialized by a \top -mapping for all program points of the method to be verified, as stated in (5.1.19a). Thus, FTC is well-defined for all program points of the method.

In order to ease readability, we begin by a sort-algebraic specification of an “code state of a code segment”. A state which consists of the first program point in the code segment, and the current frame type which the lightweight verifier associates with that program point.

Definition 5.2.2 (The Code Segment State). Let a method have the code component C and dimensions given by MS and ML . A “code state” for a code segment of C , is formally specified by

$$CST \in \text{CodeStat}_{C,MS,ML} = \text{PPoint}_C \times (\text{FrameType}_{MS,ML})_{\perp}^{\top}$$

where we omit the superscripts and subscripts from the sort names, whenever the meaning is clear from the context.

Furthermore, we simplify the lightweight verification judgment syntax by the introduction of the following verification context sort.

Definition 5.2.3 (The Lightweight Code Context). Let a method have the code component C and dimensions given by MS and ML . A “lightweight code context” for the method, is formally specified by

$$\Omega_{\text{light}} \in \text{LightContext}_{C,MS,ML} = \text{StdContext} \times \text{MethContext}_{C,MS,ML} \times \text{Cert}_{C,MS,ML}$$

where we omit the subscripts from the sort names, whenever the meaning is clear from the context.

Remark 5.2.4. Notice that the LightContext sort differs from the CodeContext sort in their third components, only, *i.e.*, FrameTypeApprox in the former sort is replaced by Cert in the latter sort.

The first rule to specify, is that of method lightweight verification which is based on the standard method verification rule (4.2.7a). Let us briefly comment on the initial and final, formal method lightweight verification requirements.

Initial constraints The initial constraints on the lightweight verifier are

- the initial pending map, P_0 ,
- the initial saved map, S_0 ,
- the initial code segment, CST_0 ,

specified in the side conditions (5.2.5a.iii) through (5.2.5a.v). The delayed constraint maps are initialized as prescribed by our formalization strategy. The initial method code segment state, is given as 0 and the initial expected frame type FT_0 . The latter being the method’s invocation frame type. The actual specification of FT_0 is commonly given with the standard verifier through the side condition (5.2.5a.vi). For further details, we refer to Definition 4.2.7. (We notice, that the actual lightweight verification of FT_0 , is performed by the code sequence rules in Definition 5.2.6.)

Final constraints The final constraints on the lightweight verifier are

- the “accumulating-condition” in (4.2.7a.x),
- the initial pending map, P_0 ,

where the ultimate requirement for a successful method verification states that the accumulated set of program points for lightweight verified instructions, must comprise all program points of the method. We notice, that the final pending map is always “garbage collected” to the initial map, as explained by the formalization strategy.

Definition 5.2.5 (Method Lightweight Verification). A method lightweight verification judgement has the signature

$$\boxed{\text{StdContext} \vdash_{\text{lbv}} \text{Method}, \text{Cert}}$$

where “ $\Gamma \vdash_{\text{lbv}} M, \text{CE}$ ” reads: the method M lightweight verifies with the certificate CE in the standard verification context Γ .

$$(5.2.5a) \quad \frac{\Omega_{\text{light}} \vdash \text{CST}_0, C, \emptyset, \langle P_0, S_0 \rangle \xrightarrow{\text{Itsafe}} \text{PPS}, \langle P_0, S \rangle}{\Gamma \vdash_{\text{lbv}} M : \text{CE}}$$

$$(5.2.5a.i) \quad \text{where } \Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$$

$$(5.2.5a.ii) \quad \text{CE} = \langle \text{FTC}, \text{LS} \rangle$$

$$(5.2.5a.iii) \quad \text{CST}_0 = \langle \emptyset, \text{FT}_0 \rangle$$

$$(5.2.5a.iv) \quad P_0 = \{ \text{PP} \mapsto \top \mid \text{PP} \in \text{PPS}_C \}$$

$$(5.2.5a.v) \quad S_0 = \{ \text{PP} \mapsto \perp \mid \text{PP} \in \text{PPS}_C \}$$

$$(5.2.5a.vi) \quad \text{same side conditions as in (4.2.7a.ii) through (4.2.7a.x).}$$

The next rule set specify the lightweight verification of a code sequence. The rules are based on the standard code-sequence verification of Rule 4.2.8a and Rule 4.2.8b. The side conditions, however, reveal how we have incorporated the lightweight specific confluence and branching properties of Figure 5.5 and Figure 5.6 into our formalizations. As we mentioned in the formalization strategy in Discussion 5.1.19, confluence properties concern all instructions of a method, whereas branching properties only concern jump instructions. Thus, we have decided to make the formal specification of the confluence-related constraints a part of the code-sequence rule, whereas the branching-related constraints are made part of the individual jump instruction verification in Section 5.3. In Figure 5.7 and Figure 5.8 we have summarized the earlier argued confluence properties, which must be met at every instruction step during code-sequence lightweight verification. As any well-defined method code-component consists of at least one instruction, code-sequence verification is given by two rules: for a single instruction in (5.2.6a), and for at least two instructions in (5.2.6b).

In order to ease the presentation of the side conditions, we have introduced the following abbreviations and optimizations with respect to Figure 5.8.

- The current frame type at PP is abbreviated to FT_{PP}^1 .
- The current frame type is generally set to $\text{FTC}(\text{PP}) \sqcap P(\text{PP}) \sqcap \text{FT}_{\text{PP}}$ regardless of the status of PP , as we observe that $P(\text{PP}) = \top$ for non-targets or backward targets. Thus, $P(\text{PP})$ does not contribute to the frame type value in those cases.
- The modification of the pending map P is generally set to $P \sqcup \{ \text{PP} \mapsto \top \}$ regardless of the status of PP , as we observe that $P(\text{PP}) = \top$ for non-targets or backward targets.
- The modification of the saved map S is generally set to $S \sqcup \{ \text{PP} \mapsto \text{FT}_{\text{PP}}^1 \}$ regardless of the status of PP . Even though the extension of S for non-targets or forward targets come as a modification with respect to Figure 5.8, we will side step this, as it does not influence on the lightweight verification process.

confluence status of PP	condition
non-target	$P(PP) = \top \wedge PP \notin LS$
backward target	$P(PP) = \top \wedge PP \in LS$
forward target	$P(PP) \neq \top \wedge PP \notin LS$
back and forward target	$P(PP) \neq \top \wedge PP \in LS$

Figure 5.7: Confluence status.

status of PP	current frame type, FT_{PP}^1	saved, S'	pending, P'
non	$FTC(PP) \sqcap FT_{PP}$	S	P
backward	$FTC(PP) \sqcap FT_{PP}$	$S \sqcup \{PP \mapsto FT_{PP}^1\}$	P
forward	$FTC(PP) \sqcap P(PP) \sqcap FT_{PP}$	S	$P \sqcup \{PP \mapsto \top\}$
back and forward	$FTC(PP) \sqcap P(PP) \sqcap FT_{PP}$	$S \sqcup \{PP \mapsto FT_{PP}^1\}$	$P \sqcup \{PP \mapsto \top\}$

Figure 5.8: Delayed type constraint updates at PP.

Definition 5.2.6 (Instruction Sequence Lightweight Verification). A code sequence, lightweight verification judgment has the signature

$$\boxed{\text{LightContext} \vdash_{\text{lbv}} \text{CodeStat}, \text{CodeSeq}, \text{PPoints}, \text{DelayConstr} \xRightarrow{\text{Itsafe}} \text{PPoints}, \text{DelayConstr}}$$

where “ $\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle, CS, PPS, DC \xRightarrow{\text{Itsafe}} PPS', DC'$ ” reads: the code sequence CS, starting at the program point PP with the frame type FT_{PP} , lightweight verifies in the context Ω_{light} on a set of delayed type safety constraints DC, and a set of lightweight verified program points PPS. The accumulated set of lightweight verified program points PPS, is extended with the program points of CS, to PPS' . The accumulated set of delayed constraints DC, is extended to DC' .

$$(5.2.6a) \quad \frac{\Omega_{\text{light}} \vdash \text{CST}', I, \langle P, S \rangle \xRightarrow{\text{Itsafe}} \langle PP', \top \rangle, P_0}{\Omega_{\text{light}} \vdash \text{CST}, I, PPS, \langle P, S \rangle \xRightarrow{\text{Itsafe}} PPS, \langle P, S \rangle}$$

$$(5.2.6a.i) \quad \text{where } \Omega_{\text{light}} = \langle -, -, CE \rangle$$

$$(5.2.6a.ii) \quad CE = \langle FTC, - \rangle$$

$$(5.2.6a.iii) \quad \text{CST} = \langle PP, FT_{PP} \rangle$$

$$(5.2.6a.iv) \quad \text{CST}' = \langle PP, FTC(PP) \sqcap P(PP) \sqcap FT_{PP} \rangle$$

$$(5.2.6a.v) \quad P_0 = P \sqcup \{PP \mapsto \top\}$$

$$(5.2.6a.vi) \quad \text{same side condition as in (4.2.8a.i).}$$

$$(5.2.6b) \quad \frac{\Omega_{\text{light}} \vdash \text{CST}', I, \langle P', S' \rangle \xRightarrow{\text{Itsafe}} \text{CST}'', P''}{\Omega_{\text{light}} \vdash \text{CST}'', C, PPS', \langle P'', S' \rangle \xRightarrow{\text{Itsafe}} PPS'', \langle P''', S'' \rangle} \\ \Omega_{\text{light}} \vdash \text{CST}, I \cdot C, PPS, \langle P, S \rangle \xRightarrow{\text{Itsafe}} PPS'', \langle P''', S'' \rangle}$$

$$\begin{aligned}
(5.2.6b.i) & \\
(5.2.6b.ii) & \quad \text{where} \quad \Omega_{\text{light}} = \langle -, -, \text{CE} \rangle \\
(5.2.6b.iii) & \quad \text{CE} = \langle \text{FTC}, - \rangle \\
(5.2.6b.iv) & \quad \text{CST} = \langle \text{PP}, \text{FT}_{\text{PP}} \rangle \\
(5.2.6b.v) & \quad \text{CST}' = \langle \text{PP}, \text{FT}_{\text{PP}}^1 \rangle \\
(5.2.6b.vi) & \quad \text{FT}_{\text{PP}}^1 = \text{FTC}(\text{PP}) \sqcap \text{P}(\text{PP}) \sqcap \text{FT}_{\text{PP}} \\
(5.2.6b.vii) & \quad \text{P}' = \text{P} \sqcup \{\text{PP} \mapsto \top\} \\
(5.2.6b.viii) & \quad \text{S}' = \text{S} \sqcup \{\text{PP} \mapsto \text{FT}_{\text{PP}}^1\} \\
(5.2.6b.ix) & \quad \text{same side conditions as in (4.2.8b.ii) through (4.2.8b.iii)}.
\end{aligned}$$

With the explanation of what we formally understand by an extension or a reduction of “saved” and “pending” in Definition 5.1.21, we observe a useful modification invariant.

Observation 5.2.7 (Delayed constraint invariants). Any lightweight verifiable JVM method and certificate, satisfy the following modification behavior.

- “Saved” is *extended or unchanged* by the lightweight sequence verification rules.
- “Pending” is *reduced or unchanged* by the lightweight sequence verification rules.

Proof. According to Remark 5.1.21, we have that the formula $\text{S} \sqcup \{\text{PP} \mapsto \text{FT}\}$ defines an extension of S and the formula $\text{P} \sqcup \{\text{PP} \mapsto \top\}$, a reduction of P . The proof follows immediately, as also reflected by Figure 5.8, since the side conditions in (5.2.6b.viii) and (5.2.6b.vii) are identical to those formulas, and the side conditions apply by the sequence rule for all instruction but the last instruction of a method, by which S and P remain unchanged. \square

Observation 5.2.8 (P invariant). The pending P is “garbage collected” in PP' by the code-sequence Rule 5.2.6b, only *after* the type constraint $\text{FTC}(\text{PP}') \sqcap \text{FT}_{\text{PP}'}^{\text{PP}}, \sqcap \text{P}(\text{PP}')$ has been established, but *before* lightweight verification of the instruction at PP' has been performed.

Finally we notice a formal property, interesting for the implementation of the lightweight procedure.

Remark 5.2.9 (Tail recursion). We notice that the proof construction for the lightweight verification of an instruction sequence, is tail recursive by the above semantics (which implies that the recursion can be efficiently implemented by a `while` loop.)

5.3 Instruction Verification

In this section, we consider each of the instruction groups which are listed in Definitions 3.1.8 through 3.1.18. The actual lightweight verification formalizations of those groups are made in parallel to the standard verification rules of Definitions 4.3.6 through 4.3.25. Generally, the only

formal difference is that the FTA component of the code context Ω has been “reduced” to CE in Ω_{light} . Something which necessitates the explicit introduction of the current, and explicit frame types into the instruction verification signatures. (In standard verification, the frame type at any program point PP is available through the application of FTA(PP)). Furthermore, as we discussed in relation to the formalization of the code-sequence rule in Definition 5.2.6, we must incorporate the branching-related data flow properties of Figure 5.6 at the individual instruction formalization level, , as these properties only concern individual jump instructions or exception handling situations. We recall Figure 5.1 for a detailed account of those jump instructions and exception situations, together with a formal summary of Figure 5.6, which describe the constraints which those jump situations must satisfy.

jump direction	constraint
$PP'' > PP$	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
$PP'' \leq PP$	$S(PP'') \sqsubseteq FT_{PP''}^{PP}$

Figure 5.9: Delayed branching constraints.

We specify the (commonly shared) lightweight instruction verification judgment signature, before we finally formalize the instruction lightweight verification.

Definition 5.3.1 (The Instruction Lightweight Verification Signature). An instruction lightweight verification judgment has the signature

$$\boxed{\text{LightContext} \vdash_{\text{lbv}} \text{CodeStat} : \text{Ins}, \text{DelayConstr} \xrightarrow{\text{Itsafe}} \text{CodeStat}, \text{Pending}}$$

where “ $\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle : I, \langle P, S \rangle \xrightarrow{\text{tsafe}} \langle PP', FT_{PP'}^{PP} \rangle, P'$ ” reads : the instruction I, at the program point PP with the current frame type FT_{PP} , lightweight verifies in the verification context Ω_{light} , on a set of delayed type constraints $\langle P, S \rangle$, with $FT_{PP'}^{PP}$, as the expected frame type for the following position PP' , and P' as the accumulated, pending set.

The lightweight verification rule for stack instructions is obtained straightforwardly from the standard verification rule in Definition 4.3.6, with the syntactic lightweight modifications described in the introduction to this section.

Definition 5.3.2 (Stack, Lightweight Instruction Verification). Take the judgment signature to be as in definition 5.3.1.

$$(5.3.2a) \quad \frac{}{\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle : I, \langle P, S \rangle \xrightarrow{\text{Itsafe}} \langle PP', FT_{PP'}^{PP} \rangle, P}$$

$$(5.3.2a.i) \quad \text{where} \quad \Omega_{\text{light}} = \langle \Gamma, \Delta, - \rangle$$

$$(5.3.2a.ii) \quad FT_{PP} = \langle ST, LT \rangle$$

$$(5.3.2a.iii) \quad \text{same side conditions as in (4.3.6a.iii) through (4.3.6a.viii).}$$

The lightweight verification rule for local variable instructions is obtained in a straightforward manner from the standard verification rule in Definition 4.3.8, with the syntactic lightweight modifications described in the introduction to this section.

Definition 5.3.3 (Local Variable, Lightweight Instruction Verification). Take the judgment signature to be as in definition 5.3.1.

(5.3.3a) same rule as in (5.3.2a).

(5.3.3a.i) where $\Omega_{\text{light}} = \langle \Gamma, \Delta, - \rangle$

(5.3.3a.ii) $\text{FT}_{\text{PP}} = \langle \text{ST}, \text{LT} \rangle$

(5.3.3a.iii) same side conditions as in (4.3.8a.iii) through (4.3.8a.ix).

The lightweight verification rule for array instructions is obtained in a straightforward manner from the standard verification rule in Definition 4.3.11, with the syntactic lightweight modifications described in the introduction to this section. According to Figure 5.1, array instructions may cause an exception to be raised, which again may lead to a jump to a handle. We postpone the formal lightweight verification of exception-caused jump situations to Section 5.4.

Definition 5.3.4 (Array, Lightweight Instruction Verification). Take the judgment signature to be as in definition 5.3.1.

$$(5.3.4a) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{tsafe}} \text{P}'}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{I}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}} \rangle, \text{P}'}$$

(5.3.4a.i) where $\Omega_{\text{light}} = \langle \Gamma, \Delta, - \rangle$

(5.3.4a.ii) $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$

(5.3.4a.iii) same side conditions as in (4.3.11a.iv) through (4.3.11a.viii).

where the premise is unfolded by the exception verification rules in Definitions 5.4.5 through 5.4.7.

The lightweight verification rule for simple access instructions is obtained in a straightforward manner from the standard verification rule in Definition 4.3.13, with the syntactic lightweight modifications described in the introduction to this section. According to Figure 5.1, simple access instructions may cause an exception to be raised, which again may lead to a jump to a handle. We postpone the formal lightweight verification of exception-caused jump situations to Section 5.4.

Definition 5.3.5 (Simple Access, Constant Pool Lightweight Verification). Take the judgment signature to be as in definition 5.3.1.

$$(5.3.5a) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{I}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}} \rangle, \text{P}'}$$

(5.3.5a.i) where $\Omega_{\text{light}} = \langle \Gamma, \Delta, - \rangle$

(5.3.5a.ii) $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$

(5.3.5a.iii) same side conditions as in (4.3.13a.iv) through (4.3.13a.ix).

where the premise is unfolded by the exception verification rules in Definitions 5.4.5 through 5.4.7.

The lightweight verification rule for field-access instructions is obtained in a straightforward manner from the standard verification rule in Definition 4.3.16, with the syntactic lightweight modifications described in the introduction to this section. According to Figure 5.1, field access instructions may cause an exception to be raised, which again may lead to a jump to a handle. We postpone the formal lightweight verification of exception-caused jump situations to Section 5.4.

Definition 5.3.6 (Field Access, Constant Pool Lightweight Verification). Take the judgment signature to be as in definition 5.3.1.

$$(5.3.6a) \quad \frac{\text{CH} \vdash T \sqsubseteq \tau \quad \Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \langle P, S \rangle \xrightarrow{\text{Itsafe}} P'}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : I, \langle P, S \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}} \rangle, P'}$$

(5.3.6a.i) where $\Omega_{\text{light}} = \langle \Gamma, \Delta, - \rangle$

(5.3.6a.ii) $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$

(5.3.6a.iii) *same side conditions as in (4.3.16a.iv) through (4.3.16a.ix).*

where the lower-most premise is unfolded by the exception verification rules in Definition 5.4.5 through Definition 5.4.7.

The lightweight verification rule for method invocation instructions is obtained straightforwardly from the standard verification rule in Definition 4.3.17, with the syntactic lightweight modifications described in the introduction to this section. According to Figure 5.1, method invocation instructions may cause an exception to be raised, which again may lead to a jump to a handle. We postpone the formal lightweight verification of exception-caused jump situations to Section 5.4.

Definition 5.3.7 (Method Invocation, Constant Pool Lightweight Verification). Take the judgment signature to be as in definition 5.3.1.

$$(5.3.7a) \quad \frac{\text{CH} \vdash T_1 \sqsubseteq \tau_1 \quad \dots \quad \text{CH} \vdash T_j \sqsubseteq \tau_j \quad \Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{ET}, \langle P, S \rangle \xrightarrow{\text{Itsafe}} P'}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : I, \langle P, S \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}} \rangle, P'}$$

(5.3.7a.i) where $\Omega_{\text{light}} = \langle \Gamma, \Delta, - \rangle$

(5.3.7a.ii) $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$

(5.3.7a.iii) *same side conditions as in (4.3.17a.iv) through (4.3.17a.ix).*

where the lower-most premise is unfolded by the exception verification rules in Definition 5.4.5 through Definition 5.4.7.

The branch instructions are jump instructions, and as such their lightweight verification rules must satisfy the delayed constraints listed in Figure 5.9. We can obtain this effect by essentially splitting the standard verification rule in Definition 4.3.20 into two rules: one for a backward jump situation, one for a forward jump situation (and with the syntactic lightweight modifications as described in the introduction to this section).

Definition 5.3.8 (Branch, Lightweight Instruction Verification). Take the judgment signature to be as in Definition 5.3.1.

$$(5.3.8a) \quad \frac{\text{CH} \vdash \mathcal{S}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}'}}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{I}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{FT}_{\text{PP}'} \rangle, \text{P}'}$$

(5.3.8a.i) where $\text{PP}'' \leq \text{PP}$

(5.3.8a.ii) $\text{FT}_{\text{PP}} = \langle \text{ST}, \text{LT} \rangle$

(5.3.8a.iii) $\Omega_{\text{light}} = \langle \Gamma, -, - \rangle$

(5.3.8a.iv) *same side conditions as in (4.3.20a.iii) through (4.3.20a.vii).*

$$(5.3.8b) \quad \frac{}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{I}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{FT}_{\text{PP}'} \rangle, \text{P}'}$$

(5.3.8b.i) where $\text{PP}'' > \text{PP}$

(5.3.8b.ii) $\text{FT}_{\text{PP}} = \langle \text{ST}, \text{LT} \rangle$

(5.3.8b.iii) $\text{P}' = \text{P} \sqcap \{ \text{PP}'' \mapsto \text{FT}_{\text{PP}'} \}$

(5.3.8b.iv) *same side conditions as in (4.3.20a.iii) through (4.3.20a.vii).*

Finally, we formalize the group of non-fall through instructions: the goto, the athrow, and the return instructions which are return, areturn, ireturn. All characterized by the fact that they, per default, impose no type requirements on the successor frame type.

The goto instruction has a special status, in that it is not only a non-fall through instruction, but also a jump instruction. As for the branch instructions, we have that the lightweight verification rules for goto must satisfy the delayed constraints, listed in Figure 5.9. An effect which can be obtained by splitting the standard verification rule in Definition 4.3.22 into two rules: one for a backward jump situation, one for a forward jump situation (and with the syntactic lightweight modifications as described in the introduction to this section.

Definition 5.3.9 (The Goto Lightweight Verification). Take the judgment signature to be as in Definition 5.3.1.

$$(5.3.9a) \quad \frac{\text{CH} \vdash \mathcal{S}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}}}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{goto}[n], \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \top \rangle, \text{P}'}$$

(5.3.9a.i) where $\text{PP}'' \leq \text{PP}$

(5.3.9a.ii) $\Omega_{\text{light}} = \langle \Gamma, -, - \rangle$

(5.3.9a.iii) *same side conditions as in (4.3.22a.iii) through (4.3.22a.v).*

$$(5.3.9b) \quad \frac{}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{goto}[n], \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \top \rangle, \text{P}'}$$

(5.3.9b.i) where $\text{PP}'' > \text{PP}$

(5.3.9b.ii) $\Omega_{\text{light}} = \langle \Gamma, -, - \rangle$

(5.3.9b.iii) $\text{P}' = \text{P} \sqcap \{ \text{PP}'' \mapsto \text{FT}_{\text{PP}} \}$

(5.3.9b.iv) *same side conditions as in (4.3.22a.iii) through (4.3.22a.v).*

The lightweight verification rule for the `athrow` instruction is obtained straightforwardly from the standard verification rule in Definition 4.3.24, with the syntactic lightweight modifications described in the introduction to this section. According to Figure 5.1, `athrow` causes an exception to be raised, which may lead to a jump to a handle. We postpone the formal lightweight verification of exception-caused jump situations to Section 5.4.

Definition 5.3.10 (The Throw Instruction Lightweight Verification). Take the judgment signature to be as in Definition 5.3.1.

$$(5.3.10a) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E, \text{ET}, \langle P, S \rangle \xrightarrow{\text{tsafe}} P'}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{athrow}, \langle P, S \rangle \xrightarrow{\text{tsafe}} \langle \text{PP}', \top \rangle, P'}$$

$$(5.3.10a.i) \quad \text{where} \quad \Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$$

$$(5.3.10a.ii) \quad \Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$$

$$(5.3.10a.iii) \quad \text{same side conditions as in (4.3.24a.iv) through (4.3.24a.viii).}$$

$$(5.3.10b) \quad \text{same rule as in (5.3.10a).}$$

$$(5.3.10b.i) \quad \text{where} \quad \Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$$

$$(5.3.10b.ii) \quad \Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{true} \rangle$$

$$(5.3.10b.iii) \quad \text{same side conditions as in (4.3.24b.iv) through (4.3.24b.viii).}$$

$$(5.3.10c) \quad \text{same rule as in (5.3.10a).}$$

$$(5.3.10c.i) \quad \text{where} \quad \Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$$

$$(5.3.10c.ii) \quad \Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$$

$$(5.3.10c.iii) \quad \text{same side conditions as in (4.3.24c.iv) through (4.3.24c.vii).}$$

where the premises are unfolded by the exception verification rules in Definitions 5.4.5 through 5.4.7.

The lightweight verification rules for the return instructions can be obtained in a straightforward manner from the standard verification rules in Definition 4.3.25, with the syntactic lightweight modifications described in the introduction to this section.

Definition 5.3.11 (The Return Instruction Lightweight Verification). Take the judgment signature to be as in Definition 5.3.1.

$$(5.3.11a) \quad \frac{}{\Omega_{\text{light}} \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle : \text{return}, \langle P, S \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \top \rangle, P}$$

(5.3.11a.i) where $\Omega_{\text{light}} = \langle -, \Delta, - \rangle$
 (5.3.11a.ii) *same side conditions as in (4.3.25a.ii) through (4.3.25a.iii).*

(5.3.11b) *same rule as in (5.3.11a).*

(5.3.11b.i) where $\Omega_{\text{light}} = \langle -, \Delta, - \rangle$
 (5.3.11b.ii) *same side conditions as in (4.3.25b.iii) through (4.3.25b.iv).*

(5.3.11c)
$$\frac{\text{CH} \vdash \tau_{\text{ob}} \sqsubseteq \tau_{\text{ob}}}{\Omega_{\text{light}} \vdash \text{PP} : \text{areturn}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \langle \text{PP}', \text{T} \rangle, \text{P}}$$

(5.3.11c.i) where $\Omega_{\text{light}} = \langle -, \Delta, - \rangle$
 (5.3.11c.ii) *same side conditions as in (4.3.25c.iii) through (4.3.25c.v).*

In parallel to Observation 5.2.7 we make the following interesting observation on instruction lightweight rules.

Observation 5.3.12 (Delayed constraint invariants). Any lightweight verifiable instruction and certificate, satisfy the following modification behavior.

- “Saved” is *unchanged* by the lightweight instruction verification rules.
- “Pending” is *extended or unchanged* by the lightweight sequence verification rules.

Proof. S is not modified by any of the lightweight instruction rules, whereas P is extended by the frame type map $\{\text{PP}'' \mapsto \text{FT}_{\text{PP}''}^{\text{PP}}\}$ over a singleton domain, which consists of the targeting program point, whenever a forward jump appears. \square

5.4 Exception Verification

In this section we discuss an approach to exception lightweight type safety and verification for the considered exception subset, which originally is specified in Definition 4.4.6. We present the formalization of this lightweight verification strategy for exceptions in Definition 5.4.4 through Definition 5.4.7. Formalizations which are created in parallel with definitions, remarks, and discussions in Section 4.4.

Discussion 5.4.1 (An Exception Lightweight Verification Strategy). The semantical impact of an exception is that it transforms the instruction which raised it into a *jump instruction*, or an *abrupt instruction*, depending on whether the exception is caught by one of the enclosing method’s exception handlers, or whether it is rethrown. In accordance with the general exception verification idea discussed in (4.4.5), the goal of a standard verification for exceptions is to assure the type safety of those semantically expected recovery patterns. According to the general lightweight

verification strategy, we must provide the same safety guarantees at the position of any handle which can possibly catch the exception, and, specifically for checked exceptions, the rethrow must be semantically well-defined for the method, if the exception is likely to pass uncaught by the exception table. In accordance with the general lightweight verification strategy, this is obtained by a two-step alterage of the standard verification semantics.

- We observe, that as for lightweight verification of jump instructions, any control transfer to an appropriate exception handler which is forward positioned in the code, will result in a modification of the accumulated pending set at that location.
- Moreover, we recall that the imposed “pending” lightweight type safety constraints, are managed by the instruction sequence rule in Definition 5.2.6. As for jump instructions, the imposed “saved” lightweight type safety constraints remains to be established by our exception formalization, whenever a catching backwards positioned handle is encountered.

These considerations lead to the decision of syntactically enriching the standard verification rules of exceptions with a delayed safety set, as well as a pending set annotation.

As the positioning of a catching handle defines the direction of the control transfer we will follow the listed procedure:

- when a handle in the exception table is likely to catch an exception in either a “definite catch” or a “potential catch”, the lightweight verification formalization splits the associated, (enriched) standard verification rule into two inference rules. One rule if the handle is backward positioned in the method, and one rule if the handle is forward positioned in the code, relative to the location where the exception can be raised.
- Specifically, if the exception may pass uncaught, lightweight verification collapses to standard verification.

We recall that the compile-time, exception “catch” concept has been formally specified in Definition 4.4.4. Furthermore, we list the notational conventions which our formalizations must adhere to.

Notation 5.4.2 (Assumptions and notational conventions). We adopt the same assumptions and notational conventions as listed in Definition 4.3.9, and Definition 5.2.1, without further modification.

In order to ease the readability of the formalizations, we introduce an abbreviated exception lightweight verification context. We notice, that the lightweight context is a simplification of the exception verification context Θ in Definition 4.4.8, though the compositions have the same semantical meanings. (In fact, only the frame type assignment component has been withdrawn.)

Definition 5.4.3 (The Exception Lightweight Verification Context).

$$\Theta_{\text{light}} \in \text{ExcContext}_{\text{light}} = \text{StdContext} \times \text{MethContext}_{\text{MS,ML}} \times \text{LocalType}_{\text{ML}} \times \text{Propagate}$$

where we omit the subscripts from the sort names, whenever the meaning is clear from the context.

Verifying rule	Lightweight rules	Catch
(4.4.12a)	(5.4.5a)	no
(4.4.12b)	(5.4.5b)	no
(4.4.12c)	(5.4.5c)	no
(4.4.12d)	(5.4.5d)	no
(4.4.12e)	(5.4.5e)	no
(4.4.14a)	(5.4.6a), (5.4.6b)	definite
(4.4.16a)	(5.4.7a), (5.4.7b)	potential

Figure 5.10: Rule correspondences for exceptions raised at PP.

The exception, lightweight verification strategy which is discussed in (5.4.1), is formalized in Definition 5.4.5 through Definition 5.4.7, with a judgment signature given by Definition 5.4.4. The formalization consists in extending the judgment syntax of the standard verification rules in Section 4.4. We recall that the standard verification rules are structured according to whether an exception is in a no-catch situation, a definite catch situation, or a potential catch situation, a structure which we will maintain in our lightweight formalization. In Figure 5.10, we have made an exhaustive listing of the lightweight rules and the, by catch property, corresponding exception verification rules. As pointed out in Discussion 5.4.1, we have furthermore split those inference rules which verifies type safety in a catch situation, into two rules. One rule for backward targeted handle positions, one rule for forward positioned handling. In Figure 6.8 (which has been placed in Chapter 6 to ease the presentation of the final proof demonstrations), we have detailed the relative position of the catching handle, and the resulting lightweight type safety measures which is taken in each case. We notice, that lightweight type safety constraints are imposed only at backward positioned handles (the third column), whereas forward positioned handling imposes what we here have called a delayed type safety verification (the fourth column).

Definition 5.4.4 (Exception Lightweight Verification Signature). An exception verification judgment signature is given as follows.

$$\text{ExcContext} \vdash_{\text{bv}} \text{PPoint}, \text{Excs}, \text{ExcHandlers}, \text{DelayConstr} \xrightarrow{\text{Itsafe}} \text{Pending}$$

where “ $\Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}$ ” reads : the set of exceptions ES, which can be raised at a program point PP, lightweight verifies on a list of exception handlers EHS on a delayed constraint set $\langle \text{P}, \text{S} \rangle$, within the lightweight context Θ_{light} , with an accumulated pending set P.

The first set of lightweight verifying rules, extends the non-catch, standard verifying inference rules in Definition 4.4.12 by a target frame type, without modifying the pending component in that proof-step.

Definition 5.4.5 (No Exception Catch). Take the judgment signature to be given as in Definition 5.4.4.

$$(5.4.5a) \quad \frac{\text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA}}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \epsilon, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}}$$

(5.4.5a.i) where $\Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, \langle -, \text{EA}, \epsilon \rangle, -, \text{PR} \rangle$

$$(5.4.5b) \quad \frac{\text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA} \quad \Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \epsilon, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}$$

(5.4.5b.i) where $\Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, \langle -, \text{EA}, \text{ET} \rangle, -, \text{PR} \rangle$

(5.4.5b.ii) *same side condition as in (4.4.12b.ii).*

$$(5.4.5c) \quad \frac{}{\Theta_{\text{light}} \vdash \text{PP}, \epsilon, \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}}$$

$$(5.4.5d) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}$$

(5.4.5d.i) *same side conditions as in (4.4.12d.i) through (4.4.12d.ii).*

$$(5.4.5e) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}$$

(5.4.5e.i) where $\Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, -, -, - \rangle$

(5.4.5e.ii) *same side conditions as in (4.4.12e.ii) through (4.4.12e.v).*

where the top premises of (5.4.5a) and (5.4.5b), which lightweight verifies CID_E as an uncaught exception, collapses with standard verification of an uncaught exception, already specified in Definition 4.4.19.

The next set of lightweight verifying rules, splits and extends the standard verifying rule in (4.4.14) into two rules: one which establishes lightweight type safety at a definite, backward positioned handle, and one which introduce a delayed safety constraint for a definite, forward positioned handle. Thus, only the “forward situation” causes the pending set to be modified. As in (4.4.14), the remaining, unverified exceptions are lightweight verified with the exception table, in a separate premise.

Definition 5.4.6 (Definite Exception Catch). Take the judgment signature to be given as in Definition 5.4.4.

$$(5.4.6a) \quad \frac{\text{CH} \vdash S(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}} \quad \Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{ET}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}''}$$

$$(5.4.6a.i) \quad \text{where } \text{PP}'' \leq \text{PP}$$

$$(5.4.6a.ii) \quad \Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, \langle -, -, \text{ET} \rangle, \text{LT}, - \rangle$$

$$(5.4.6a.iii) \quad \text{same side conditions as in (4.4.14a.iii) through (4.4.14a.vi).}$$

$$(5.4.6b) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{ET}, \langle \text{P}', \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}''}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}''}$$

$$(5.4.6b.i) \quad \text{where } \text{PP}'' > \text{PP}$$

$$(5.4.6b.ii) \quad \Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, \langle -, -, \text{ET} \rangle, \text{LT}, - \rangle$$

$$(5.4.6b.iii) \quad \text{P}' = \text{P} \sqcap \{\text{PP}'' \mapsto \text{FT}_{\text{PP}''}^{\text{PP}}\}$$

$$(5.4.6b.iv) \quad \text{same side conditions as in (4.4.14a.iii) through (4.4.14a.vi).}$$

The final set of lightweight verifying rules, splits and extends the standard verifying rule in (4.4.16) into two rules: one which establishes lightweight type safety at a potential, backward positioned handle, and one which introduce a delayed safety constraint for the a potential, forward positioned handle. Thus, only the “forward situation” causes the pending set to be modified. As in (4.4.16), the current exception is furthermore lightweight verified with the remaining exception handles, in a separate premise.

Definition 5.4.7 (Potential Catch Situation). Take the judgment signature to be given as in Definition 5.4.4.

$$(5.4.7a) \quad \frac{\text{CH} \vdash S(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}} \quad \Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}'}$$

$$(5.4.7a.i) \quad \text{where } \text{PP}'' \leq \text{PP}$$

$$(5.4.7a.ii) \quad \Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, -, \text{LT}, - \rangle$$

$$(5.4.7a.iii) \quad \text{same side conditions as in (4.4.16a.iii) through (4.4.16a.vi).}$$

$$(5.4.7b) \quad \frac{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \langle \text{P}', \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}''}{\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \langle \text{P}, \text{S} \rangle \xrightarrow{\text{Itsafe}} \text{P}''}$$

- (5.4.7b.i) where $PP'' > PP$
 (5.4.7b.ii) $\Theta_{\text{light}} = \langle \langle -, \text{CH} \rangle, -, \text{LT}, - \rangle$
 (5.4.7b.iii) $P' = P \sqcap \{PP'' \mapsto \text{FT}_{PP''}^{PP}\}$
 (5.4.7b.iv) *same side conditions as in (4.4.16a.iii) through (4.4.16a.vi).*

Observation 5.4.8 (lightweight verification of uncaught exceptions). A static analysis of whether a checked, user-thrown exception can be rethrown from a method, comes back to a type check of the method's exception attribute. Because this type check is independent of jumps in the code, it does not impose any lightweight safety constraints, or contribute to the pending component; and as such, collapses with the already formalized verification in Definition 4.4.19 and Definition 4.4.20.

Remark 5.4.9 (Error lightweight verification). Since `Error` describes a class of unchecked exceptions, these can be lightweight verified by the presented formalizations, however, under the same constraints as explained in Discussion 4.4.22.

5.5 The Example

We present a lightweight verification proof-unfolding for our `cksum()` method, based on the formalization of its verification context, frame type approximation, and class hierarchy, all of which are summarized in Figure 3.8 and Figure 4.6.

Proposition 5.5.1 (cksum lightweight verifies). The `cksum()` method M^{ck} , lightweight verifies in the context Γ^{ck} , *i.e.*, $\Gamma^{\text{ck}} \vdash_{\text{bv}} M^{\text{ck}}, CE^{\text{ck}}$, with the certificate:

$$(5.5.1a) \quad CE^{\text{ck}} = \langle \text{FTC}^{\text{ck}}, \text{vls}^{\text{ck}} \rangle$$

$$(5.5.1b) \quad \text{FTC}^{\text{ck}} = \emptyset$$

$$(5.5.1c) \quad \text{LS}^{\text{ck}} = \{20\}$$

Proof. Assume Γ^{ck} , M^{ck} , and CE^{ck} as in the Proposition. Then the lightweight rules of this chapter can be used to construct the proof

$$(5.5.2) \quad \frac{\frac{\boxed{\text{A.2.1}}}{\Omega_{\text{light}}^{\text{ck}} \vdash \langle 0, \text{FT}_0^{\text{ck}} \rangle, C^{\text{ck}}, \emptyset, \langle P_0^{\text{ck}}, S_0^{\text{ck}} \rangle \xRightarrow{\text{Itsafe}} \text{PPS}^{\text{ck}}, \langle P_0^{\text{ck}}, S_{20}^{\text{ck}} \rangle} (5.2.6b)}{\Gamma^{\text{ck}} \vdash M^{\text{ck}}, CE^{\text{ck}}} (5.2.5a)}{\Gamma^{\text{ck}} \vdash M^{\text{ck}}, CE^{\text{ck}}} (5.2.5a)$$

where

$$(5.5.2.i) \quad \Omega_{\text{light}}^{\text{ck}} = \langle \Gamma^{\text{ck}}, \Delta^{\text{ck}}, CE^{\text{ck}} \rangle$$

$$(5.5.2.ii) \quad \Delta^{\text{ck}} = \langle \text{int}, \text{MFR}^{\text{ck}}, \text{EA}^{\text{ck}}, \text{ET}^{\text{ck}} \rangle$$

$$(5.5.2.iii) \quad P_0^{\text{ck}} = \{PP \mapsto \top \mid PP \in \text{PPS}_C\}$$

$$(5.5.2.iv) \quad S_0^{\text{ck}} = \{PP \mapsto \perp \mid PP \in \text{PPS}_C\}$$

$$(5.5.2.v) \quad S_{20}^{\text{ck}} = \{PP \mapsto \perp \mid PP \in \text{PPS}_C\} \sqcup \{20 \mapsto \text{FTA}(20)\}$$

and where $\boxed{\text{A.2.1}}$ denotes the full proof tree given in Appendix A.2. □

Chapter 6

Lightweight Certification Formalization

In this chapter, we semantically formalize a lightweight certificate for type safe bytecode as a synergy of the verification specifications from Chapters 4 and 5 and use the certification formalization to show that the lightweight verification of the method with such a certificate establishes the same type safety guarantees as standard bytecode verification (for our instruction subset).

In Section 6.1, we discuss how to construct and formalize a lightweight certificate for a type safe method bytecode, and what to understand by “certification”. Then, in Section 6.2, we recall our formalization approach and specify the certification of method bytecode accordingly. Also, we state our main theorem, which shows that lightweight bytecode verification provides the same type safety guarantees as standard bytecode verification. In Section 6.3, we semantically specify certification for our instruction subset. Finally, in Section 6.5, we present a complete certification proof unfolding for our canonical checksum method.

6.1 Analysis and Formalization Strategy

The design of a certificate is closely associated with both lightweight verification, as specified in Chapter 5, and standard verification, as specified in Chapter 4. In this section, we specify the formal requirements to a certificate, based on the role it plays in lightweight verification. Furthermore, we discuss how such a certificate can be obtained from a standard method verification, followed by a formal presentation of a certification system. Our goal is to prove our main theorem relating the standard and lightweight bytecode verification systems.

Theorem 6.1.1 (lightweight bytecode verification is safe). *For any standard verification context Γ and method M the following statements are equivalent:*

1. *There exists a frame type assignment FTA such that $\Gamma \vdash_{bv} M, FTA$ is provable.*
2. *There exists a lightweight certificate CE such that $\Gamma \vdash_{lbv} M, CE$ is provable.*

Proof. Follows directly from the properties of the certification system to be shown in Lemma 6.2.6 below. \square

Discussion 6.1.2 (A lightweight certification strategy). Recall that the intent with a certificate is to be designed in such a way that the lightweight verifier can reconstruct a solution to the method’s

constraint set in one straight code-pass, provided the method is verifiable. In Figure 5.2, we informally listed a set a necessary requirements to the design of a certificate, which lead to a sort algebraic specification of a certificate in Definition 5.1.7. A specification which composes a certificate as a set of backward jump targets LS and a frame type map FTC . Let us discuss each of those components.

According to the *lightweight assumption*, we have that it is possible to construct a solution to a verifiable method’s constraint set from an appropriate certificate in one, straight code pass. The lightweight procedure, which we at first described naively in Example 5.1.15, suggests that the solution at a backward jump target PP is recorded for a later constraint check, as the lightweight procedure eventually reaches the jump. In Proposition 5.1.17, we formalized the constraint which must be satisfied for PP , that is

$$(6.1.2a) \quad CH \vdash S(PP) \sqsubseteq FT_{PP}^{PP^{(+j)}}, \quad PP \leq PP^{(+j)}$$

Clearly, this constraint-check cannot be made during one, straight code pass, unless we *know whether a program point is a backward label*. Thus, the set of backward labels for a method is a necessary component to include in a certificate. Specifically for the canonical example method $cksum()$, LS will contain the backward label 20.

According to Proposition 5.1.16, we can characterize the current frame type at a program point PP during a straight code pass, from the forward jumps which is targeting PP , and the expected frame type from the previous instruction. However, it is not possible to take into account the constraints imposed by backward jumps to PP at the time when the lightweight procedure reaches PP , not even if the backward targets are known in advance. Thus, the current frame type in PP can at most be characterized as a “naive” solution to a verifiable method’s constraint set, as a naive lightweight procedure will construct it without knowing anything else (than the backward labels). With the assumptions and notation given in Proposition 5.1.16, such a “naive” solution is formalized by

$$(6.1.2b) \quad CH \vdash \prod_{i=1}^k P(PP)_{PP^{(-i)}} \sqcap FT_{PP}^{PP^{(-1)}}, \quad PP > PP^{(-i)}$$

As we discussed in (5.1.19), the problem for the naive lightweight approach is that the procedure fails to proceed after a non-fall through instruction, which only is targeted by a backward jump. In this case, the so called “naive” lightweight solution in that program point is formally given by the expected frame type, which becomes the trivial frame type \top . A simple remedy to make the lightweight procedure pass such a program point PP of a verifiable method, would be to provide a frame type solution $FTA(PP)$ externally, by the certificate. But as Example 6.1.3 shows, the solution we can reconstruct for $cksum()$, by a lightweight verifier as specified in Chapter 5, cannot be expected to be identical to the solution from which the certificate provided it’s frame type component.

One could object that it does not matter, as long as we can state that the method verifies. However, it complicates the equivalence proofs between lightweight verification and standard verification. In this formalization, we therefore aim at a certificate design which not only accomplishes the lightweight assumption for verifiable methods, but which in doing so, automatically reconstructs an isomorphic frame type solution to the method’s constraint set as the one from which the certificate was deduced with our design principles.

Our strategy to achieve this goal is to discuss how such an “isomorphic” certificate design must look like, and to systematically list the principles after which such a certificate must be deduced by our formalization. Along with the presentations of the resulting certification system, we will then simultaneously prove the equivalence of lightweight verification and standard verification. A proof which not only justifies the chosen certificate design, but also establishes the desired equivalence of type safe programs shown by standard verification, and type safe programs shown by lightweight verification.

Let us consider a certificate for the solution in Figure 6.1 of Example 6.1.3. If we only included the smaller solution’s frame types at the program points which follows a non-fall through instruction, *i.e.*, 10, 15, 36, and 47, the naive lightweight procedure would have been able to re-construct a solution, but not the more restricted solution FTA' from the figure. The problem is, that whenever a frame type solution can be given, which at a program point is “less defined” in the associated method frame-type lattice than the current frame type in that program point, we have *no constructive way* to achieve that solution again based on such a restricted set of frame types. One way to remedy this, is to include *all* of those frame types where the solution differs from the current frame type which the naive lightweight procedure can find. That is whenever

$$(6.1.2c) \quad CH \vdash FTA'(PP) \sqsubset \prod_{i=1}^k P(PP)_{PP^{(-i)}} \sqcap FT_{PP}^{PP^{(-1)}}, \quad PP > PP^{(-i)}$$

For the solution FTA' in Figure 6.1, this means that the frame type component of the certificate must be given by

$$(6.1.2d) \quad \begin{aligned} FTC' = \{ & 0 \mapsto FTA'(0), 5 \mapsto FTA'(5), \\ & 10 \mapsto FTA'(10), 32 \mapsto FTA'(32), \\ & 41 \mapsto FTA'(41), 44 \mapsto FTA'(44), \\ & 49 \mapsto FTA'(49), 53 \mapsto FTA'(53) \} \end{aligned}$$

as the constraint

$$(6.1.2e) \quad CH \vdash FTA'(PP) \sqsubset \left(\prod_{i=1}^{k_{PP}} P(PP)_{PP^{(-i)}} \sqcap FT_{PP}^{PP^{(-1)}} \right)$$

is satisfied for all $PP \in \{0, 5, 10, 32, 41, 44, 49, 53\}$ where $k_{10} = 2$, otherwise $k_{PP} = 1$. (Recall that the initial, expected frame type $FT_0^{0^{(-1)}}$ is defined as FT_0 .)

In comparison, we have that for the solution FTA in Figure 4.5 of Example 4.5, the frame type component, deduced by the same certificate principles, becomes

$$(6.1.2f) \quad FTC = \emptyset$$

We notice, that the deduced label set LS for the code associated with the solution FTA' in Example 6.1, is the same as for FTA in Example 6.1, as the code sequence of the `cksum()` method has not changed between the two examples. Thus, the difference in certificate size for different solutions, only a concern to the frame type certificate component. The general discussion on the theoretical

PP	FTA' _{pp} .ST	FTA' _{pp} .LT (this.ccnum·x·y·z)	I
0	ε	⊥·CrCardRd·⊥·⊥·⊥	aload[1]
2	CrCardRd	⊥·CrCardRd·⊥·⊥·⊥	invokevirtual[1]
5	int	⊥·⊥·⊥·⊥·⊥	istore[2]
7	ε	⊥·⊥·int·⊥·⊥	goto[+8]
10	⊥	⊥·⊥·⊥·⊥·⊥	pop
11	ε	⊥·⊥·⊥·⊥·⊥	new[2]
14	Abort	⊥·⊥·⊥·⊥·⊥	athrow
15	ε	⊥·⊥·int·⊥·⊥	ldc.w[3]
18	int	⊥·⊥·int·⊥·⊥	istore[3]
20	ε	⊥·⊥·int·int·⊥	iload[2]
22	int	⊥·⊥·int·int·⊥	iload[3]
24	int·int	⊥·⊥·int·int·⊥	isub
25	int	⊥·⊥·int·int·⊥	istore[4]
27	ε	⊥·⊥·int·int·int	iload[4]
29	int	⊥·⊥·int·int·int	ifle[+10]
32	ε	⊥·⊥·⊥·int·int	iload[4]
34	int	⊥·⊥·⊥·int·int	istore[2]
36	ε	⊥·⊥·int·int·int	goto[-16]
39	ε	⊥·⊥·int·int·int	iload[4]
41	int	⊥·⊥·int·int·⊥	ifne[+6]
44	ε	⊥·⊥·int·⊥·⊥	iload[2]
46	int	⊥·⊥·⊥·⊥·⊥	ireturn
47	ε	⊥·⊥·int·int·⊥	iload[2]
49	int	⊥·⊥·⊥·int·⊥	istore[4]
51	ε	⊥·⊥·⊥·int·int	iload[3]
53	int	⊥·⊥·⊥·⊥·int	istore[2]
55	ε	⊥·⊥·int·⊥·int	iload[4]
57	int	⊥·⊥·int·⊥·⊥	istore[3]
59	ε	⊥·⊥·int·int·⊥	goto[-39]

Figure 6.1: A “smaller” cksum() frame type assignment.

relationship between the certificate size and the solution by which the certificate is deduced, however, is postponed to Chapter 8. For the time being, we will formally list the discussed certificate design principles in Definition 6.1.4, before we proceed by a discussion on how to formalize a system of those criteria.

Example 6.1.3 (A smaller frame type solution). The $\text{cksum}()$ method with a code component C , which we showed to bytecode verify earlier, could have verified on a less defined FTA. A “smaller” solution to FTA is a solution FTA' to the method’s constraint set, such that $\text{FTA}'(\text{PP}) \sqsubseteq \text{FTA}(\text{PP})$ for all $\text{PP} \in \text{PPS}_C$. In Figure 6.1, we give an example of such a frame typing.

Based on Discussion 6.1.2, we summarize the design principles by which our certification system shall specify a lightweight certificate.

Definition 6.1.4 (A Lightweight Certificate Design). Let C be a code component of some verifiable method, and FTA a solution to the method’s constraint set. Let the assumptions be as in Proposition 5.1.16.

1. If the instruction at PP is a jump instruction and PP'' is the jump target, we have that if $\text{PP}'' \leq \text{PP}$ then $\text{PP}'' \in \text{LS}$ by our certification strategy.
2. Whenever $\text{CH} \vdash \text{FTA}(\text{PP}) \sqsubset \left(\prod_{i=1}^k \text{P}(\text{PP})_{\text{PP}(-i)} \sqcap \text{FT}_{\text{PP}}^{\text{PP}(-1)} \right)$, we have that FTC is extended with FTA in PP .

Discussion 6.1.5 (The formalization strategy). The concept of a certificate is based on the notion of a (standard) verifiable solution to a method’s constraint set, according to the Discussion 6.1.2. Thus, it seems natural that a formal certification specification is achieved from the standard bytecode verification system in Chapter 4. Specifically, with the certificate design requirements in Definition 6.1.4, a lightweight certificate can be seen as a “logical property” of standard bytecode verification. Consequently, a formalization can be achieved by an extension of the standard verification system with syntactic certificate annotations, such that a certificate becomes given as an accumulated set of *backward labels* and *frame type mappings*, which is recursively defined through sub-rules.

First we notice, that the two design requirements in Definition 6.1.4, expresses two types of constraints as already discussed in Section 5.1. The *confluence-related constraints* in this context concern the identification of a program point’s pending properties, syntactically given by P , in order to construct the correct, current frame type, whereas the *branching related-constraints* in this context, concern the identification of backward jump targets for jump instructions and exception handling situations. The former constraints serve to extend the frame type certificate, syntactically given by FTC , appropriately, whereas the latter constraints serve to extend the backward label set, syntactically given by LS . As argued by the formalization of lightweight verification, confluence constraints must be applied to all program points and as such should be specified by the most general (sequencing) certification rules, whereas branching constraints are properties of individual instruction certification. Let us briefly list how this affects the (re)organization of the standard verification rules.

- In accordance with the second design principle in Definition 6.1.4, we must split each general rule into two, in order to extend the frame type certificate FTC appropriately. Thus, one rule if the frame type solution $\text{FTA}(\text{PP})$ is less defined than the current frame type $\text{CH} \vdash \prod_{i=1}^k \text{P}(\text{PP})_{\text{PP}(-i)} \sqcap \text{FT}_{\text{PP}}^{\text{PP}(-1)}$ in that program point, another one if they are identical.
- In accordance with the lightweight verification of instructions in Section 5.3, we must split each of the jump instruction, or exception handling rules into two, in order to extend both the backward label set LS, and the pending set P appropriately.

Finally we notice, that we will adopt the same initialization map for P as in Definition 5.2.5, and a similar initialization of FTC.

$$(6.1.5a) \quad \text{FTC}_0 = \{\text{PP} \mapsto \top \mid \text{PP} \in \text{PPS}_C\}$$

6.2 Bytecode Certification

The goal of certification is to deduce a certificate (at the code generating platform), given a standard verified method, which is sufficient to re-verify the type safety of the method by lightweight verification (upon arrival at the code destination platform). In this sense, certification can be seen as an extension of standard bytecode verification. Thus, we specify the certification rules in natural semantics [22], as syntactic (and semantic) extensions of the standard verification semantics, with a certificate. Furthermore, we restate the type-safety equivalence Theorem 6.1.1 in more detail as a lemma, and formulate its proof.

Notation 6.2.1 (Assumptions and notational conventions). We adopt the following considerations in our formalizations.

- Same assumptions and notational conventions as listed for lightweight (and standard) verification in Notation 5.2.1
- We assume that a frame type certificate FTC, for a method code component C, is initialized by the map $\{\text{PP} \mapsto \top \mid \text{PP} \in \text{PPS}_C\}$.
- We introduce the following abbreviations for each of the three formal verification semantics in this thesis. Here specified by their principal judgement.

$$\begin{aligned} (\text{BV}) \quad & \Gamma \vdash_{\text{bv}} M, \text{FTA} \quad (\text{Standard bytecode verification}) \\ (\text{LBV}) \quad & \Gamma \vdash_{\text{lbv}} M, \text{CE} \quad (\text{Lightweight bytecode verification}) \\ (\text{LBC}) \quad & \Gamma \vdash_{\text{lbc}} M, \text{FTA}, \text{CE} \quad (\text{Lightweight bytecode certification}) \end{aligned}$$

In order to ease the readability of the formalizations, we introduce a sort-algebraic extension of the certificate sort, which is specified as follows.

Definition 6.2.2 (The Pending Certificate Sort). Let a method be given by the code component C and dimensions given by MS and ML. A “pending certificate” for the method is formally specified by

$$(6.2.2a) \quad \text{PCE} \in \text{PCert}_{C, \text{MS}, \text{ML}} = \text{Pending}_{C, \text{MS}, \text{ML}} \times \text{Cert}_{C, \text{MS}, \text{ML}}$$

Remark 6.2.3. We have called it a “pending certificate sort”, because it enriches the certificate with forward jump details along the accumulation of the certificate. The term “pending certificate” explicit that a pending map P is built along with any certification judgement proof unfolding.

The formalization of method certification, is given as a syntactic (and semantic) extension of the standard method verification Judgement 4.2.7a, in accordance with Discussion 6.1.2, and with Discussion 6.1.5. Let us briefly glance over the initial and final requirements to the formalization of method certification.

Initial constraints The initial constraints on the certifier are

- the initial pending map is P_0 ,
- the initial label set is \emptyset ,
- the initial frame type certificate is FTC_0 ,

as specified by the side conditions (6.2.4a.i) and (6.2.4a.ii), and by the side conditions (6.2.4b.ii) and (6.2.4b.iii). The initial pending map P_0 and the initial frame type certificate FTC_0 , are both initialized in accordance with Discussion 6.1.5. The initially expected frame type is given by the method’s invocation frame type FT_0 . The specification of FT_0 is in common with that of standard verification by the side condition (5.2.5a.vi). Finally we notice, that the certification of FT_0 is performed by the sequence rule in Definition 5.2.6.

Final constraints the final constraints on the lightweight verifier are

- the “accumulating-condition” is specified by (4.2.7a.x),
- the final pending map is P_0 (the initial),

where the ultimate requirement for a successful method certification states that the accumulated set of program points PPS' of certified instructions, must comprise all program points in the method. We notice, that the final pending map is always “garbage collected” to the initial map, as explained in Discussion 6.1.5.

Definition 6.2.4 (Method Certification). A method bytecode certification judgement has the signature

$$\boxed{\text{StdContext} \vdash_{\text{lbc}} \text{Method}, \text{FrameTypeApprox}, \text{Cert}}$$

where “ $\Gamma \vdash_{\text{lbc}} M, \text{FTA}, \text{CE}$ ” reads: the method code M bytecode- certifies with a frame type assignment FTA in the verification context Γ , with an accumulating certificate CE .

$$(6.2.4a) \quad \frac{\text{CH} \vdash \text{FTA}(0) = FT_0 \quad \Omega \vdash \emptyset, C, \emptyset, \text{PCE}_0 \xRightarrow{\text{certify}} \text{PPS}, \text{PCE}}{\Gamma \vdash_{\text{lbc}} M, \text{FTA}, \text{CE}}$$

$$(6.2.4a.i) \quad \text{where} \quad \text{FTC}_0 = \{ \text{PP} \mapsto \top \mid \text{PP} \in \text{PPS}_C \}$$

$$(6.2.4a.ii) \quad P_0 = \{ \text{PP} \mapsto \top \mid \text{PP} \in \text{PPS}_C \}$$

- (6.2.4a.iii) $PCE_0 = \langle P_0, \langle FTC_0, \emptyset \rangle \rangle$
 (6.2.4a.iv) $PCE = \langle P_0, CE \rangle$
 (6.2.4a.v) *same side conditions as in (4.2.7a.i) through (4.2.7a.x).*

$$(6.2.4b) \quad \frac{CH \vdash FTA(0) \sqsubseteq FT_0 \quad \Omega \vdash 0, C, \emptyset, PCE_0 \xrightarrow{\text{certify}} PPS, PCE}{\Gamma \vdash_{\text{lbc}} M, FTA, CE}$$

- (6.2.4b.i) *where* $FTC_1 = FTC_0 \sqcap \{0 \mapsto FTA(0)\}$
 (6.2.4b.ii) $FTC_0 = \{PP \mapsto \top \mid PP \in PPS_C\}$
 (6.2.4b.iii) $P_0 = \{PP \mapsto \top \mid PP \in PPS_C\}$
 (6.2.4b.iv) $PCE_0 = \langle P_0, \langle FTC_1, \emptyset \rangle \rangle$
 (6.2.4b.v) $PCE = \langle P_0, CE \rangle$
 (6.2.4b.vi) *same side conditions as in (4.2.7a.i) through (4.2.7a.x).*

Remark 6.2.5. We notice, how the standard verification rule in (4.2.7a) has been split into two, almost identical rules in Definition 6.2.4. The rules are mutual exclusive, in that the uppermost premises distinguish between an “equally defined” and a “less defined” frame type assignment case in program point 0 (as proposed in Discussion 6.1.2).

We restate Theorem 6.1.1 as a lemma which relates (BV), (LBV), and (LBC).

Lemma 6.2.6 (BV-LBC and LBC-LBV equivalences). Within the verification context Γ , we have that if a method M standard verifies successfully with a frametype approximation FTA , then there exists a uniquely given certificate CE by which the method certifies, and vice versa.

$$(6.2.6a) \quad \Gamma \vdash_{\text{bv}} M, FTA \Leftrightarrow \exists! CE : \Gamma \vdash_{\text{lbc}} M, FTA, CE$$

Furthermore we have that within the verification context Γ , if a method M can be lightweight verified with a given certificate CE , then CE is unique in assisting M to be certified with some frame type assignment FTA , and vice versa.

$$(6.2.6b) \quad \Gamma \vdash_{\text{lbv}} M, CE \Leftrightarrow \exists! FTA : \Gamma \vdash_{\text{lbc}} M, FTA, CE$$

Proof. We prove each implication of either bi-implication case separately. The proofs of the arguments are based on showing that given the assumed proof tree we can construct the implied proof tree.

Case $BV \Rightarrow LBC$. Assume a BV proof tree with (4.2.7a) at the base and subproofs for $CH \vdash FTA(0) \sqsubseteq FT_0$ and $\Omega \vdash 0, C, \emptyset \xrightarrow{\text{tsafe}} PPS$. We can identify the former as either a subproof for frame type equality $CH \vdash FTA(0) = FT_0$ or frame type reduction $CH \vdash FTA(0) \sqsubseteq FT_0$. From Lemma 6.2.14 we will have a subproof for $\Omega \vdash 0, C, \emptyset, PCE_0 \xrightarrow{\text{certify}} PPS, PCE$ defined by the side conditions of either (6.2.4a) for the frame type equality case or (6.2.4b) for the frame type reduction case, and we know that the constructed certificate is unique. This now constitutes a proof of $\exists! CE : \Gamma \vdash_{\text{lbc}} M, FTA, CE$ with either (6.2.4a) or (6.2.4b) as the base rule.

Case BV \Leftarrow LBC. Assume an LBC proof tree with either (6.2.4a) or (6.2.4b) at the base. By (4.1.28f) or (4.1.28g), and from Lemma 6.2.14 we will have subproofs for $\text{CH} \vdash \text{FTA}(0) \sqsubseteq \text{FT}_0$ and $\Omega \vdash 0, C, \emptyset \xRightarrow{\text{tsafe}} \text{PPS}$ which means that we have a proof of BV with (4.2.7a) at the base by erasing all LBC-only sort components.

Case LBV \Rightarrow LBC. Assume an LBV proof tree with the rule of Definition 5.2.5 at the base and a subproof for $\Omega_{\text{light}} \vdash \text{CST}_0, C, \emptyset, \langle P_0, S_0 \rangle \xRightarrow{\text{Itsafe}} \text{PPS}, \langle P_0, S \rangle$ From Lemma 6.2.14 we will have a proof of $\Omega \vdash 0, C, \emptyset, \text{PCE}_0 \xRightarrow{\text{certify}} \text{PPS}, \text{PCE}$ where one of the following two situations hold: either $\text{CH} \vdash \text{FTA}(0) = \text{FT}_0$ is provable and the side conditions of (6.2.4a) hold, or $\text{CH} \vdash \text{FTA}(0) \sqsubseteq \text{FT}_0$ is provable and the side conditions of (6.2.4b) hold. In either case the result constitutes an LBC proof with (4.2.7a) at the base.

Case LBV \Leftarrow LBC. Assume an LBC proof tree with either (6.2.4a) or (6.2.4b) at the base. With $\Omega_{\text{light}}, \text{CST}_0$, and S_0 defined as in the side conditions of Definition 5.2.5 we then by Lemma 6.2.14 will have a subproof for $\Omega_{\text{light}} \vdash \text{CST}_0, C, \emptyset, \langle P_0, S_0 \rangle \xRightarrow{\text{Itsafe}} \text{PPS}, \langle P_0, S \rangle$ which means that we have a proof of $\Gamma \vdash_{\text{lbv}} M, \text{CE}$.

The four cases combine to prove the two equivalences. \square

In order to ease the readability of the code-sequence certification rules, we introduce the following sort-algebraic definition.

Definition 6.2.7 (The Pre-Delayed Constraint Sort). Let a method have the code component C and dimensions given by MS and ML. A “pre-delayed constraint” for the method is formally specified by

$$(6.2.7a) \quad \text{PCT} \in \text{PreConstr}_{C, \text{MS}, \text{ML}} = \text{Pending}_{C, \text{MS}, \text{ML}} \times \text{Labels}_C$$

where we omit the subscripts from the sort names, whenever the meaning is clear from the context.

Remark 6.2.8. We have called it a “pre-delayed constraint sort”, because of the similarity with the delayed frame type constraint sort DelayConstr from Definition 5.1.10. In fact, the difference comes with the second component of DelayConstr in that the “saved” map component, S, is replaced by an accumulated set of backward labels, LS, in a pre-constraint delayed set PCT. (The LS set defines the non-trivial domain subset of the saved map¹ during lightweight verification.)

The next rule set specifies lightweight certification of a code sequence. The rules are based on standard verification of a code sequence in rules (4.2.8a) and (4.2.8b).

Definition 6.2.9 (Instruction Sequence Certification). A code sequence certification judgment has the signature

$$\boxed{\text{CodeContext} \vdash_{\text{lbc}} \text{PPoint}, \text{CodeSeq}, \text{PPoints}, \text{PCert} \xRightarrow{\text{certify}} \text{PPoints}, \text{PCert}}$$

¹Both pending and saved are defined as total maps.

where “ $\Omega \vdash PP, CS, PPS, PCE \xRightarrow{\text{certify}} PPS', PCE'$ ” reads : the code sequence CS, starting at program point PP on a set of certified program points PPS and a pending-certificate PCE, is certified in the code verification context Ω with an accumulated set of certified program points PPS' , and an accumulated, pending-certificate PCE' .

$$(6.2.9a) \quad \frac{\Omega \vdash PP, I, PCT \xrightarrow{\text{certify}} PP', \top, PCT_0}{\Omega \vdash PP, I, PPS, PCE \xRightarrow{\text{certify}} PPS', PCE'}$$

$$(6.2.9a.i) \quad PCE = \langle P, CE \rangle$$

$$(6.2.9a.ii) \quad PCE' = \langle P_0, CE \rangle$$

$$(6.2.9a.iii) \quad CE = \langle -, LS \rangle$$

$$(6.2.9a.iv) \quad PCT = \langle P, LS \rangle$$

$$(6.2.9a.v) \quad PCT_0 = \langle P_0, LS \rangle$$

$$(6.2.9a.vi) \quad P_0 = P \sqcup \{PP \mapsto \top\}$$

$$(6.2.9a.vii) \quad \text{same side condition as in (4.2.8a.i).}$$

$$(6.2.9b) \quad \frac{\begin{array}{l} CH \vdash FTA(PP') = FT_{PP'}^1 \\ \Omega \vdash PP : I_1, PCT \xrightarrow{\text{certify}} PP', FT_{PP'}^{PP}, PCT' \\ \Omega \vdash PP', I_2 \cdot CS, PPS', PCE' \xRightarrow{\text{certify}} PPS'', PCE'' \end{array}}{\Omega \vdash PP, I_1 \cdot I_2 \cdot CS, PPS, PCE \xRightarrow{\text{certify}} PPS'', PCE''}$$

$$(6.2.9b.i) \quad \text{where } \Omega = \langle \Gamma, -, FTA \rangle$$

$$(6.2.9b.ii) \quad \Gamma = \langle -, CH \rangle$$

$$(6.2.9b.iii) \quad CE = \langle LS, FTC \rangle$$

$$(6.2.9b.iv) \quad CE' = \langle LS', FTC \rangle$$

$$(6.2.9b.v) \quad PCE = \langle P, CE \rangle$$

$$(6.2.9b.vi) \quad PCE' = \langle P'', CE' \rangle$$

$$(6.2.9b.vii) \quad PCT = \langle P', LS \rangle$$

$$(6.2.9b.viii) \quad PCT' = \langle P'', LS' \rangle$$

$$(6.2.9b.ix) \quad FT_{PP'}^1 = FT_{PP'}^{PP} \sqcap P(PP')$$

$$(6.2.9b.x) \quad P' = P \sqcup \{PP \mapsto \top\}$$

$$(6.2.9b.xi) \quad \text{same side conditions as in (4.2.8b.i) through (4.2.8b.iii).}$$

$$\begin{array}{l}
\text{CH} \vdash \text{FTA}(\text{PP}') \sqsubseteq \text{FT}_{\text{PP}'}^1 \\
\Omega \vdash \text{PP} : I_1, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}, \text{PCT}' \\
\Omega \vdash \text{PP}', I_2 \cdot \text{CS}, \text{PPS}', \text{PCE}' \xrightarrow{\text{certify}} \text{PPS}'', \text{PCE}'' \\
\hline
\Omega \vdash \text{PP}, I_1 \cdot I_2 \cdot \text{CS}, \text{PPS}, \text{PCE} \xrightarrow{\text{certify}} \text{PPS}'', \text{PCE}''
\end{array}
\tag{6.2.9c}$$

$$\begin{array}{ll}
(6.2.9c.i) & \text{where } \Omega = \langle \Gamma, -, \text{FTA} \rangle \\
(6.2.9c.ii) & \Gamma = \langle -, \text{CH} \rangle \\
(6.2.9c.iii) & \text{CE} = \langle \text{LS}, \text{FTC} \rangle \\
(6.2.9c.iv) & \text{CE}' = \langle \text{LS}', \text{FTC}' \rangle \\
(6.2.9c.v) & \text{PCE} = \langle \text{P}, \text{CE} \rangle \\
(6.2.9c.vi) & \text{PCE}' = \langle \text{P}'', \text{CE}' \rangle \\
(6.2.9c.vii) & \text{PCT} = \langle \text{P}', \text{LS} \rangle \\
(6.2.9c.viii) & \text{PCT}' = \langle \text{P}'', \text{LS}' \rangle \\
(6.2.9c.ix) & \text{FT}_{\text{PP}'}^1 = \text{FT}_{\text{PP}'}^{\text{PP}} \sqcap \text{P}(\text{PP}') \\
(6.2.9c.x) & \text{FTC}' = \text{FTC} \sqcap \{ \text{PP}' \mapsto \text{FTA}(\text{PP}') \} \\
(6.2.9c.xi) & \text{P}' = \text{P} \sqcup \{ \text{PP} \mapsto \top \} \\
(6.2.9c.xii) & \text{same side conditions as in (4.2.8b.i) through (4.2.8b.iii)}.
\end{array}$$

Remark 6.2.10. We notice how the standard verification rule in (4.2.8b) has been split into two, almost identical rules in (6.2.9b) and (6.2.9c). The rules are mutual exclusive, in that the uppermost premises distinguish between an “equally defined” and a “less defined” frame type assignment case in the subsequent program point, following the first instruction in the code sequence. In accordance with Discussion 6.1.5, the frame type certificate FTC is accumulated in the latter case, as specified in (6.2.9c.xi).

Notation 6.2.11. The single-instruction rule in (6.2.9a), implies a slight abuse of notation, since PP' cannot be the program point of an empty code segment. In accordance with earlier remarks, however, we side-step this fact as the program point PP' in (6.2.9a), merely has a symbolic status as it is skipped by the conclusion. Thus, by abuse of notation, we tacitly assume that $\text{PP}' \in \text{PPoints}$.

Observation 6.2.12 (Pending Constraints Invariant). The pending P is “garbage collected” in PP' by the code-sequence Rule 6.2.9c, only *after* the type constraint $\text{CH} \vdash \text{FTA}(\text{PP}') \sqsubseteq \text{FT}_{\text{PP}'}^{\text{PP}} \sqcap \text{P}(\text{PP}')$ has been established, but *before* certification of the instruction at PP' has been performed.

We formulate an assumption which is invariant in the meaning of PPS , LS , P , S , PP and FTA in our code sequence formalizations, based on the observation that all of our semantics are tail recursive in the code.

Definition 6.2.13 (Program Point Consistency). We assume that PPS , LS , P , S , PP , and FTA are given as specified in our verification formalizations.

1. PPS and LS are *consistent* with PP if all members are smaller than PP.
2. P is *consistent* with PP and FTA if $PP' \leq PP$ implies $P(PP') = \top$ and $PP' > PP$ implies $FTA(PP') \sqsubseteq P(PP')$.
3. S is *consistent* with PP and FTA if $PP' < PP$ implies $S(PP') = FTA(PP)$ and $PP' \geq PP$ implies $S(PP') = \perp$.
4. FTC is *consistent* with PP and FTA if $PP' \leq PP$ implies $FTC(PP') = FTA(PP')$ and $PP' > PP$ implies $FTC(PP') = \top$.
5. A composite sort element is *consistent* with PP and FTA if all of its constituents are consistent with PP and FTA.

Lemma 6.2.14 (Sequence Verification Equivalences). Let PPS be the collection of all program points in the full code and P_0 be the initial and final pending set. The following are equivalent for given $\Gamma, \Delta, FTA, PP, C$, and PPS' , with PPS' consistent with PP:

- $\langle \Gamma, \Delta, FTA \rangle \vdash_{bv} PP, C, PPS' \stackrel{tsafe}{\Rightarrow} PPS$, and
- Find $\langle P', CE' \rangle$ and PPS' consistent with PP and FTA and a unique CE such that

$$(6.2.14a) \quad \langle \Gamma, \Delta, FTA \rangle \vdash_{lbc} PP, C, PPS', \langle P', CE' \rangle \stackrel{certify}{\Rightarrow} PPS, \langle P_0, CE \rangle$$

Conversely, the following are equivalent for given Γ, Δ, PP, C , and P' and PPS' consistent with PP:

- Find S' consistent with PP and FTA and S consistent with $\max(PPS)$ such that

$$(6.2.14b) \quad \langle \Omega, \Delta, CE \rangle \vdash_{lbc} \langle PP, FT \rangle, C, PPS', \langle P', S' \rangle \stackrel{tsafe}{\Rightarrow} PPS, \langle P_0, S \rangle$$

- Find FTA and CE' consistent with PP and FTA such that

$$(6.2.14c) \quad \langle \Gamma, \Delta, FTA \rangle \vdash_{lbc} PP, C, PPS', \langle P', CE' \rangle \stackrel{certify}{\Rightarrow} PPS, \langle P_0, CE \rangle$$

Proof sketch. Each of the four cases is an induction over the existing proof tree with the step showing how to construct the base inference of the desired implied proof tree. In each case the assumptions suffice to construct it and even show that when constructing a CE from FTA (or vice versa) then the result is unique. It is crucial that the rules are all “tail recursive” thus having the *final* CE, P_0 , or S, (as appropriate) present in each step. \square

6.3 Instruction Certification

We have formalized each instruction group, which have been specified in Definition 3.1.8 through Definition 3.1.18, separately. As pointed out earlier, the certification rules are obtained as an extension of the standard verification rules in Definition 4.3.6 through Definition 4.3.25, in accordance with the formalization certification strategy described in Discussion 6.1.5.

We begin our presentation with a sort-algebraic specification of the instruction certification judgement signature, common for all instruction groups.

Definition 6.3.1 (The Instruction Certification Signature). An instruction certification judgment has the signature

$$\boxed{\text{CodeContext} \vdash_{\text{lbc}} \text{PPoint} : \text{Ins}, \text{PreConstr} \xrightarrow{\text{certify}} \text{PPoint}, \text{FrameType}, \text{PreConstr}}$$

where “ $\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{PCT}', \text{FT}_{\text{PP}'}$ ” reads: the instruction I , at the program point PP , certify in the code verification context Ω , on a set of pre-delayed constraints PCT , with the expected frame type $\text{FT}_{\text{PP}'}$, in the successor point PP' , and the accumulated pre-delayed constraint set PCT' .

The certification rule for stack instructions can be obtained straight forwardly as a trivial, syntactic extension of the standard verification rule in Definition 4.3.6.

Definition 6.3.2 (Stack Instruction Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.2a) \quad \frac{}{\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}, \text{PCT}}$$

(6.3.2a.i) *same side conditions as in (4.3.6a.i) through (4.3.6a.viii).*

The certification rule for local variable instructions can be obtained straight forwardly as a trivial, syntactic extension of the standard verification rule in Definition 4.3.8.

Definition 6.3.3 (Local Variable Instruction Certification).

(6.3.3a) *same rule as in (6.3.2a).*

(6.3.3a.i) *same side conditions as in (4.3.8a.i) through (4.3.8a.ix).*

The lightweight certification rule for array instructions is obtained in a straightforward manner as a trivial, syntactic extension of the standard verification rule in Definition 4.3.11. As array instructions may cause an exception to be raised, which again may lead to a jump to a handle, the pre-constraint pair is accumulated by an exception premise which we specify further in Section 6.4.

Definition 6.3.4 (Array Instruction Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.4a) \quad \frac{\Theta \vdash \text{PP}, \text{ES}, \text{ET}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}{\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}, \text{PCT}'}$$

(6.3.4a.i) where $\text{PCT} = \langle \text{P}, \text{LS} \rangle$

(6.3.4a.ii) $\text{PCT}' = \langle \text{P}', \text{LS}' \rangle$

(6.3.4a.iii) *same side conditions as in (4.3.11a.i) through (4.3.11a.viii).*

The lightweight certification rule for simple-access instructions is obtained in a straightforward manner as a trivial, syntactic extension of the standard verification rule in Definition 4.3.13. As simple-access instructions may cause an exception to be raised, which again may lead to a jump to a handle, the pre-constraint pair is accumulated by an exception premise which we specify further in Section 6.4.

Definition 6.3.5 (Simple Access, Constant Pool Instruction Certification).

(6.3.5a) *same rule as in (6.3.4a).*

(6.3.5a.i) *same side conditions as in (4.3.13a.i) through (4.3.13a.ix).*

The lightweight certification rule for field-access instructions is obtained in a straightforward manner as a trivial, syntactic extension of the standard verification rule in Definition 4.3.16. As field-access instructions may cause an exception to be raised, which again may lead to a jump to a handle, the pre-constraint pair is accumulated by an exception premise which we specify further in Section 6.4.

Definition 6.3.6 (Field Access, Constant Pool Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.6a) \quad \frac{\text{CH} \vdash T \sqsubseteq \tau \quad \Theta \vdash \text{PP}, \text{ES}, \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}{\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}, \text{PCT}'}$$

(6.3.6a.i) where $\text{PCT} = \langle \text{P}, \text{LS} \rangle$

(6.3.6a.ii) $\text{PCT}' = \langle \text{P}', \text{LS}' \rangle$

(6.3.6a.iii) *same side conditions as in (4.3.16a.i) through (4.3.16a.ix).*

The lightweight certification rule for the method invocation instructions is obtained in a straightforward manner as a syntactic extension of the rule in Definition 4.3.17. As method invocation instructions may cause an exception to be raised, which again may lead to a jump to a handle, the pre-delayed constraint pair is accumulated in a premise, which we specify further in Section 6.4.

Definition 6.3.7 (Method Invocation, Constant Pool Instruction Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.7a) \quad \frac{\text{CH} \vdash T_1 \sqsubseteq \tau_1 \quad \dots \quad \text{CH} \vdash T_j \sqsubseteq \tau_j \quad \Theta \vdash \text{PP}, \text{ES}, \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}{\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}, \text{PCT}'}$$

(6.3.7a.i) *same side conditions as in (4.3.17a.i) through (4.3.17a.ix).*

The branch instructions are jump instructions, and as such their lightweight certification rules must accumulate the backward label set for backward directed jumps, and extend the pending map for forward directed jumps. As discussed, we obtain this by splitting the standard verification rule in Definition 4.3.20 into two rules: one for a backward jump situation, one for a forward jump situation. Each one being a trivial, syntactic extension of the standard verification rule in Definition 4.3.17.

Definition 6.3.8 (Branch Instruction Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.8a) \quad \frac{\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}}{\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}, \text{PCT}'}$$

$$(6.3.8a.i) \quad \text{where} \quad \text{PP}'' \leq \text{PP}$$

$$(6.3.8a.ii) \quad \text{PCT} = \langle \text{P}, \text{LS} \rangle$$

$$(6.3.8a.iii) \quad \text{PCT}' = \langle \text{P}, \text{LS}' \rangle$$

$$(6.3.8a.iv) \quad \text{LS}' = \text{LS} \cup \{\text{PP}''\}$$

$$(6.3.8a.v) \quad \text{same side conditions as in (4.3.20a.i) through (4.3.20a.vii).}$$

$$(6.3.8b) \quad \frac{\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}}{\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}'}^{\text{PP}}, \text{PCT}'}$$

$$(6.3.8b.i) \quad \text{where} \quad \text{PP}'' > \text{PP}$$

$$(6.3.8b.ii) \quad \text{PCT} = \langle \text{P}, \text{LS} \rangle$$

$$(6.3.8b.iii) \quad \text{PCT}' = \langle \text{P}', \text{LS} \rangle$$

$$(6.3.8b.iv) \quad \text{P}' = \text{P} \sqcap \{\text{PP}'' \mapsto \text{FT}_{\text{PP}''}^{\text{PP}}\}$$

$$(6.3.8b.v) \quad \text{same side conditions as in (4.3.20a.i) through (4.3.20a.vii).}$$

The goto instruction is a jump instructions, and as such we have that the associated lightweight certification rules must accumulate the backward label set for a backward directed jump, and extend the pending map for a forward directed jump. We obtain this by splitting the standard verification rule in Definition 4.3.22 into two rules: one for a backward jump situation, one for a forward jump situation. Each one being a trivial, syntactic extension of the standard verification rule in Definition 4.3.22.

Definition 6.3.9 (Goto Instruction Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.9a) \quad \frac{\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}}{\Omega \vdash \text{PP} : \text{goto}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \top, \text{PCT}'}$$

$$(6.3.9a.i) \quad \text{where} \quad \text{PP}'' \leq \text{PP}$$

- (6.3.9a.ii) $PCT = \langle P, LS \rangle$
 (6.3.9a.iii) $PCT' = \langle P, LS' \rangle$
 (6.3.9a.iv) $LS' = LS \cup \{PP''\}$
 (6.3.9a.v) *same side-conditions as in (4.3.22a.i) through (4.3.22a.v).*

$$(6.3.9b) \quad \frac{CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP}}{\Omega \vdash PP : \text{goto}, PCT \xrightarrow{\text{certify}} PP', \top, PCT'}$$

- (6.3.9b.i) where $PP'' > PP$
 (6.3.9b.ii) $PCT = \langle P, LS \rangle$
 (6.3.9b.iii) $PCT' = \langle P', LS \rangle$
 (6.3.9b.iv) $P' = P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
 (6.3.9b.v) *same side-conditions as in (4.3.22a.i) through (4.3.22a.v).*

The lightweight certification rule for the `athrow` instruction is obtained straightforwardly as a trivial extension of the standard verification rules in Definition 4.3.24. As `athrow` causes an exception to be raised, the pre-constraint pair is accumulated by an exception premise which we specify further in Section 6.4.

Definition 6.3.10 (Throw Instruction Certification). Take the judgment signature to be as in Definition 6.3.1.

$$(6.3.10a) \quad \frac{\Theta \vdash PP, CID_E, EHS, PCT \rightarrow PCT'}{\Omega \vdash PP : \text{athrow}, PCT \xrightarrow{\text{certify}} PP', \top, PCT'}$$

- (6.3.10a.i) *same side conditions as in (4.3.24a.i) through (4.3.24a.viii).*

- (6.3.10b) *same rule as in (4.3.24a).*

- (6.3.10b.i) *same side conditions as in (4.3.24b.i) through (4.3.24b.viii).*

- (6.3.10c) *same rule as in (4.3.24a).*

- (6.3.10c.i) *same side conditions as in (4.3.24c.i) through (4.3.24c.vii).*

The lightweight certification rules for return instructions are finally obtained as trivial extensions of the standard verification rules in Definition 4.3.25.

Definition 6.3.11 (Return Instruction Certification). Take the judgement signature to be as in Definition 6.3.1.

$$(6.3.11a) \quad \frac{}{\Omega \vdash PP : \text{return}, PCT \xrightarrow{\text{certify}} PP', \top, PCT}$$

(6.3.11a.i) *same side-conditions as in (4.3.25a.i) through (4.3.25a.iii).*

$$(6.3.11b) \quad \frac{}{\Omega \vdash PP : \text{ireturn}, PCT \xrightarrow{\text{certify}} PP', \top, PCT}$$

(6.3.11b.i) *same side-conditions as in (4.3.25b.i) through (4.3.25b.iv).*

$$(6.3.11c) \quad \frac{CH \vdash T_{ob} \sqsubseteq \tau_{ob}}{\Omega \vdash PP : \text{areturn}, PCT \xrightarrow{\text{certify}} PP', \top, PCT}$$

(6.3.11c.i) *same side-conditions as in (4.3.25c.i) through (4.3.25c.v).*

We proceed by proving, at first, some invariants and equivalence properties between standard verification rules in (BV) and certification rules in (LBC).

Observation 6.3.12 (Side condition invariant). Any bound side condition in a standard method, sequence, and instruction verification rule is also a bound side condition in the corresponding method, sequence, and instruction certification rule. Consequently, the same symbols, functions, and abbreviations can be assumed to have the same meaning in the two method, sequence, and instruction systems.

Proof. The observation follows directly from the side condition listings in Definition 6.2.4 through Definition 6.3.11, and the fact that the formal certification system comes as a pure syntactic extension of the standard verification system. \square

Observation 6.3.13 (Type constraint invariants). The type safety constraints which are marked (BV-invar) and (LBC-invar) are established at each program point of any well-defined method².

For any program point PP with PP' as the successor program point, standard verification rules impose the following constraint for a verifiable method with a solution FTA:

$$(6.3.13a) \quad CH \vdash \text{FTA}(PP') \sqsubseteq \text{FT}_{PP'}^{PP} \quad (\text{BV-invar})$$

whereas the lightweight certification rules, however, impose the following constraint for the method's program points:

$$(6.3.13b) \quad CH \vdash \text{FTA}(PP') \sqsubseteq \text{FT}_{PP'}^{PP} \sqcap P(PP') \quad (\text{LBC-invar})$$

By observation 6.3.12, we can assume that the frame types in the two constraints means the same thing as they are given by the same notation.

²A method code sequence contains at least one instruction.

Instructions	Verifier rule	Certifier rules	Type constraint
stack	(4.3.6a)	(6.3.2a)	—
local variable	(4.3.8a)	(6.3.3a)	—
array	(4.3.11a)	(6.3.4a)	Proposition 6.4.10
simple	(4.3.13a)	(6.3.5a)	Proposition 6.4.10
field	(4.3.16a)	(6.3.6a)	Proposition 6.4.10
method	(4.3.17a)	(6.3.7a)	Proposition 6.4.10
branch	(4.3.20a)	(6.3.8a), (6.3.8b)	$\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$
goto	(4.3.22a)	(6.3.9a), (6.3.9b)	$\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$
athrow	(4.3.24a)	(6.3.10a)	Proposition 6.4.10
athrow	(4.3.24b)	(6.3.10b)	Proposition 6.4.10
athrow	(4.3.24c)	(6.3.10c)	Proposition 6.4.10
return	(4.3.25a)	(6.3.11a)	—
ireturn	(4.3.25b)	(6.3.11b)	—
areturn	(4.3.25c)	(6.3.11c)	—

Figure 6.2: Instruction rule type safety.

Proof. We reason over the set of program points by considering two separate cases.

1. $\text{PP}' = 0$. The invariants are explicitly given as premises by the method rules in Definition 4.2.7 and Definition 6.2.4.
2. $\text{PP}' > 0$. Easily seen, as (BV-invar) is recursively given for all program points as a premise of the standard sequence verification rule in Definition 4.2.8, and as the (LBC-invar) is recursively given for all program points as a premise of the sequence certification rule in Definition 6.2.9.

□

Observation 6.3.14 (P invariant). We observe, that

- $\text{P}(\text{PP}) = \top$ at any proof-step if PP is not a jump target. In fact, this property follows from the fact that P may only be extended by forward jump instruction rules, or forward jump exception-rules, in Figure 6.4 and Figure 6.7.
- $\text{P}(\text{PP}) \neq \top$ By the same argument, we now that there is at least one forward target to PP .

Specifically we have that, as 0 is never a forward target, $\text{P}(0) = \top$ at any proof-step.

Proposition 6.3.15. A method which satisfies the (BV-invar) in (BV) satisfies the (LBC-invar) in (LBC), and vice versa.

Proof. We reason over the set of program points by considering two separate cases.

Rule	Handling at PP''	Label update	Pending update
(6.3.2a)	—	—	—
(6.3.3a)	—	—	—
(6.3.4a)	—	Figure 6.7	Figure 6.7
(6.3.5a)	—	Figure 6.7	Figure 6.7
(6.3.6a)	—	Figure 6.7	Figure 6.7
(6.3.7a)	—	Figure 6.7	Figure 6.7
(6.3.8a)	backwards	$LS \cup \{PP''\}$	—
(6.3.8b)	forwards	—	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
(6.3.9a)	backwards	$LS \cup \{PP''\}$	—
(6.3.9b)	forwards	—	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
(6.3.10a)	—	Figure 6.7	Figure 6.7
(6.3.10b)	—	Figure 6.7	Figure 6.7
(6.3.10c)	—	Figure 6.7	Figure 6.7
(6.3.11a)	—	—	—
(6.3.11b)	—	—	—
(6.3.11c)	—	—	—

Figure 6.3: Instruction certification updates.

Case $PP' = 0$. (LBC-invar) \Rightarrow (BV-invar) According to Observation 6.3.14, $P(0) = \top$. Thus, by the lattice property of the frame type order, we have that $CH \vdash FTA(0) = FT_0 \sqcap P(0)$ implies $CH \vdash FTA(0) = FT_0$, and $CH \vdash FTA(0) \sqsubseteq FT_0 \sqcap P(0)$ implies $CH \sqsubseteq FTA(0) \sqsubseteq FT_0$.

(BV-invar) \Rightarrow (LBC-invar) By the lattice property of the frame type order, we can make the trivial extension $FT_{PP'}^{PP} = FT_{PP'}^{PP} \sqcap \top$, Thus, by Observation 6.3.14, we immediately have $CH \vdash FTA(0) \sqsubseteq FT_0 \sqcap P(0)$.

Case $PP' > 0$. (LBC-invar) \Rightarrow (BV-invar) Follows immediately from Lemma 6.3.19.

(BV-invar) \Rightarrow (LBC-invar) According to Lemma 6.3.22 we have

$$(6.3.16) \quad CH \vdash FTA(PP') \sqsubseteq FT_{PP'}^{PP}$$

$$(6.3.17) \quad CH \vdash FTA(PP') \sqsubseteq P(PP')$$

and consequently in (LBC) that

$$(6.3.18) \quad CH \vdash FTA(PP') \sqsubseteq FT_{PP'}^{PP} \sqcap P(PP')$$

□

Lemma 6.3.19. Let a verifiable method with a solution FTA be given, and let PP, PP'' be two arbitrary program points of that method. When the constraint $CH \vdash FTA(PP'') \sqsubseteq P(PP'')$ is established in (LBC), $CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP}$ is established in (BV).

Proof. When an (LBC)-rule establishes the constraint

$$(6.3.20) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}} \sqcap \text{P}(\text{PP}')$$

for arbitrary program points PP , PP'' , we have by the definition of \sqcap that

$$(6.3.21) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$$

Exception raising instructions From Figure 6.2 we observe that an exception raising rule in (BV) correspond to one exception raising rule in (LBC) for the same instruction. Specifically we have, according to Proposition 6.4.10, that these rules establish the same type constraints in the same program points.

Jump instructions and non-exception raising instructions From Figure 6.2 we observe that when the constraint $\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$ is established for a jump target PP'' in (LBC), the same constraint is established in (BV) (either by (4.3.20a) or by (4.3.22a)). Consequently, (6.3.21) is also a constraint of (BV). □

Lemma 6.3.22. Let a verifiable method with a solution FTA be given, and let PP, PP'' be two arbitrary program points of that method. When the constraint $\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$ is established in (BV), then $\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{P}(\text{PP}'')$ is established in (LBC).

Proof. We reason over the set of instructions by considering two separate cases.

Exception raising instructions From Figure 6.2 we observe that an exception raising rule in (BV) correspond to one exception raising rule in (LBC) for the same instruction. Specifically, we have, according to Proposition 6.4.10, that these rules establish the same type constraints in the same program points.

Jump instructions and non-exception raising instructions From Figure 6.2 we see that when the constraint $\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$ is established for a target PP'' in (BV), the same constraint is established in (LBC) (either by (6.3.8a) or (6.3.8b), or by (6.3.9a) or (6.3.9b)). Assume that PP'' is a forward jump which is targeted from k jump instructions in the method, say, positioned at $\text{PP}^{(-1)}, \text{PP}^{(-2)}, \dots, \text{PP}^{(-k)}$. According to Figure 6.2, we have

$$(6.3.23) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}^{(-1)}}$$

$$(6.3.24) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}^{(-2)}}$$

$$(6.3.25) \quad \dots\dots$$

$$(6.3.26) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}^{(-k)}}$$

By the lattice property of the frame type ordering follows that

$$(6.3.27) \quad \text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \prod_{j=1}^k \text{FT}_{\text{PP}''}^{\text{PP}^{(-j)}}$$

is established both in (BV) and in (LBC).

According to Figure 6.3, P is extended in (LBC) when PP'' is a forward jump target. Specifically, P is extended with a singleton map to the imposed frame type at PP'' . By generalization we have that

$$(6.3.28) \quad \prod_{j=1}^k \text{FT}_{PP''}^{PP^{(-j)}} = \left(\prod_{j=1}^k P(PP'')_{PP^{(-j)}} \right) = P(PP'')$$

and consequently that

$$(6.3.29) \quad \text{CH} \vdash \text{FTA}(PP'') \sqsubseteq P(PP'')$$

is established in (LBC).

Finally we have that according to Observation 6.3.14, $P(PP'') = \top$ in (LBC) if PP'' is *not a forward jump target*. Thus, (6.3.29) is trivially satisfied for all program points of the method. \square

Now we show some invariant and equivalence properties between the certification rules in (LBC) and the lightweight verification rules in (LBV).

Observation 6.3.30 (Side condition invariant). Just as is the case for the inductive Lemma 6.2.14, most of the side conditions which originates from (BV) are also side conditions of the lightweight verification rules. Consequently, the symbols, functions, and abbreviations which are bound by rules in all three of those systems can be assumed to have the same meaning in either.

Proof. The observation follows directly from the side condition listings in Definition 5.2.5 through Definition 5.4.7, and the fact that the formal lightweight system, apart from “FTA” being replaced by “ce”, is a syntactic extension of the verification system. We notice, that the listed symbols which differs between the two systems are bound *either* by side conditions of (LBC) rules *or* by side conditions of (LBV) rules. Thus, with a semantic meaning which is given by either (LBC) or (LBV). \square

Based on the Observation 5.3.12 on the lightweight instruction verification rules in Chapter 5, we obtain the Figure 6.4.

We begin by showing two important properties on type constraints and updates.

Proposition 6.3.31. For any program point PP and PP'' of some verifiable method, we have that both (LBC) and (LBV) update P with the same frame type in the same program points.

Proof. We reason over the instruction subset by dividing the proof after the following instruction partitions.

Exception raising instructions. Postponed to Proposition 6.4.11.

Jump instructions and non-exception raising instructions From the listings in Figure 6.3, Figure 6.4, and Figure 6.5 we observe that P is updated by the same frame type, by forward jump instruction-rules which correspond one-to-one between the two verification systems.

Rule	Handling at PP''	Saved constraint	Pending update
(5.3.2a)	—	—	—
(5.3.3a)	—	—	—
(5.3.4a)	—	Figure 6.8	Figure 6.8
(5.3.5a)	—	Figure 6.8	Figure 6.8
(5.3.6a)	—	Figure 6.8	Figure 6.8
(5.3.7a)	—	Figure 6.8	Figure 6.8
(5.3.8a)	backwards	$CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$	—
(5.3.8b)	forwards	—	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
(5.3.9a)	backwards	$CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$	—
(5.3.9b)	forwards	—	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
(5.3.10a)	—	Figure 6.8	Figure 6.8
(5.3.10b)	—	Figure 6.8	Figure 6.8
(5.3.10c)	—	Figure 6.8	Figure 6.8
(5.3.11a)	—	—	—
(5.3.11b)	—	—	—
(5.3.11c)	—	—	—

Figure 6.4: Delayed type safety checks and updates.

□

We generalize Proposition 6.3.31 to include the following result.

Corollary 6.3.32 (Pending invariant). *The pending maps in (LBC) and (LBV) contain the same frame type elements at the corresponding proof-steps.*

Proof. We observe, that P is initialized by the \top -mapping at all method program points at the first proof step in both (LBC) and (LBV). From Observation 6.2.12, Observation 5.2.8 and Proposition 6.4.11 we conclude that P is “garbage collected” and updated in some program point in the same sequence³ in (LBC) as well as in (LBV). □

Proposition 6.3.33. For any program point PP and PP'' of some verifiable method, there is a one-to-one correspondence between the rules which check the saved constraint in (LBV), that is

$$(6.3.33a) \quad CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$$

and the rules which update the backward label set in (LBC), that is

$$(6.3.33b) \quad LS \cup \{PP''\}$$

Proof. We reason over the instruction subset by dividing the proof after the following instruction partitions.

Exception raising instructions. Postponed to Proposition 6.4.12.

³ P is “garbage collected in PP by application of $P \sqcap \{PP \mapsto \top\}$.”

Instruction group	stack	locals	array	simple	field	method
Certifying rule	(6.3.2a)	(6.3.3a)	(6.3.4a)	(6.3.5a)	(6.3.6a)	(6.3.7a)
Lightweight rule	(5.3.2a)	(5.3.3a)	(5.3.4a)	(5.3.5a)	(5.3.6a)	(5.3.7a)

Instruction group	branch	branch	goto	goto	throw
Certifying rule	(6.3.8a)	(6.3.8b)	(6.3.9a)	(6.3.9b)	(6.3.10a)
Lightweight rule	(5.3.8a)	(5.3.8b)	(5.3.9a)	(5.3.9b)	(5.3.10a)

Instruction group	throw	throw	return	ireturn	areturn
Certifying rule	(6.3.10b)	(6.3.10c)	(6.3.11a)	(6.3.11b)	(6.3.11c)
Lightweight rule	(5.3.10b)	(5.3.10c)	(5.3.11a)	(5.3.11b)	(5.3.11c)

Figure 6.5: Instruction rule correspondence.

Jump instructions and non-exception raising instructions From the listings in Figure 6.3, Figure 6.4, and Figure 6.5 we can observe that LS is extended by the same program point PP'' in (LBC) whenever the constraint

$$(6.3.34) \quad CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$$

is checked in (LBV), by backward jump instruction-rules which correspond one-to-one between the two verification systems (that is, whenever PP'' is a backward jump target from PP).

□

6.4 Exception Certification

In this section we begin by a discussion of what it means to certify an exception for the considered exception subset, which originally is specified in Definition 4.4.6, followed by the actual formalization of exception certification in Definition 6.4.4 through Definition 6.4.6. Furthermore we present some invariants and equivalence properties for (BV), (LBV), and (LBC) exception rules, used above.

Discussion 6.4.1 (An Exception Certification Strategy). In accordance with the general strategy, we will base our exception certification on the standard verification rules for exceptions from Section 4.4. Exceptions are special with respect to certification in that they represent pure jump situations whenever they can be handled by the exception table. Thus causes an update of the the method's jump set in any jump catch situation. Our formalization strategy is therefore to divide any verification rule into two rules in the certification specification: one rule for when the handle is backwards located, one rule for when it is forwards located.

We recall the exception-specific concepts introduced in Section 4.4 and summarize the notational conventions we have adopted.

Notation 6.4.2 (Assumptions and conventions). We adopt the same assumptions and notational conventions as listed in Definition 4.2.5 and Definition 4.3.4, as well as the formal exception context which was introduced in Definition 4.4.8.

The actual exception certification is formalized in Definition 6.4.4 through Definition 6.4.6, with a judgment signature given by Definition 6.4.3. The formalization consists in extending the judgment syntax of the standard verification rules in Section 4.4. We recall that the standard verification rules are structured according to whether an exception is in a no-catch jump situation, a definite jump catch situation, or a potential jump catch situation. A structure which it makes sense to maintain for certification.

Definition 6.4.3 (The Exception Certification Signature). A judgment which certifies an exception comes with the following signature.

$$\boxed{\text{ExcContext} \vdash_{\text{lbc}} \text{PPoint}, \text{ExcS}, \text{ExcHandlers}, \text{PreConstr} \xrightarrow{\text{certify}} \text{PreConstr}}$$

where “ $\Theta \vdash \text{PP}, \text{EHS}, \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'$ ” reads: the set of compile-time exceptions, ES, which may be raised at the program point PP, verifies within the exception verification context Θ , on a list of the exception table handlers EHS and on a pre-constraint set PCT, with an accumulated pre-constraint set PCT'.

The first set of certification rules extends the standard verifying inference rules in (4.4.12a) through (4.4.12e), without, as expected, contributing to the jump set during the unfolding step.

Definition 6.4.4 (No Exception Catch). Take the judgment signature to be given as in Definition 6.4.3.

$$(6.4.4a) \quad \frac{\text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA}}{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}}$$

(6.4.4a.i) *same side conditions as in (4.4.12a.i).*

$$(6.4.4b) \quad \frac{\begin{array}{c} \text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA} \\ \Theta \vdash \text{PP}, \text{ES}, \text{ET}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}' \end{array}}{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \epsilon, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}$$

(6.4.4b.i) *same side conditions as in (4.4.12b.i) through (4.4.12b.ii).*

$$(6.4.4c) \quad \frac{}{\Theta \vdash \text{PP}, \epsilon, \text{EHS}, \text{LS} \xrightarrow{\text{certify}} \text{LS}}$$

$$(6.4.4d) \quad \frac{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}$$

(6.4.4d.i) *same side conditions as in (4.4.12d.i) through (4.4.12d.ii).*

$$(6.4.4e) \quad \frac{\Theta \vdash PP, CID_E \cdot ES, EHS, PCT \xrightarrow{\text{certify}} PCT'}{\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS, PCT \xrightarrow{\text{certify}} PCT'}$$

(6.4.4e.i) *same side conditions as in (4.4.12e.i) through (4.4.12e.v).*

The next set of certification rules extends the standard verifying inference rule in (4.4.14a) that is a definite jump catch situation. Clearly we have to split the verification rule into two parallel rules, according to the jump direction, and accumulate the jump set.

Definition 6.4.5 (Definite Exception Catch). Take the judgment signature to be given as in Definition 6.4.3.

$$(6.4.5a) \quad \frac{\begin{array}{c} CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP} \\ \Theta \vdash PP, ES, ET, PCT \xrightarrow{\text{certify}} PCT' \end{array}}{\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS, PCT \xrightarrow{\text{certify}} PCT''}$$

$$(6.4.5a.i) \quad \text{where } PCT' = \langle P', LS' \rangle$$

$$(6.4.5a.ii) \quad PCT'' = \langle P', LS'' \rangle$$

$$(6.4.5a.iii) \quad LS'' = LS' \cup \{PP''\}$$

$$(6.4.5a.iv) \quad PP'' \leq PP$$

(6.4.5a.v) *same side conditions as in (4.4.14a.i) through (4.4.14a.vi).*

(6.4.5b) *same rule as in (6.4.5a).*

$$(6.4.5b.i) \quad \text{where } PCT' = \langle P', LS' \rangle$$

$$(6.4.5b.ii) \quad PCT'' = \langle P'', LS' \rangle$$

$$(6.4.5b.iii) \quad PP'' > PP$$

$$(6.4.5b.iv) \quad P'' = P' \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$$

(6.4.5b.v) *same side-conditions as in (4.4.14a.i) to (4.4.14a.vi).*

The final set of certification rules extends the standard verifying inference rule in (4.4.16a) that is a potential catch situation. The certification is identical to that of definite catch certification. The difference is to find in the proof trees of the two situations. Whereas the jump set is accumulated over the remaining raised exceptions in a definite catch situation, it is, in addition, being accumulated for the current exception over the remaining exception table in a potential catch situation.

Definition 6.4.6 (Potential Exception Catch). Take the judgment signature to be given as in Definition 6.4.3.

$$(6.4.6a) \quad \frac{\text{CH} \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}} \quad \Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'}{\Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}''}$$

$$(6.4.6a.i) \quad \text{where} \quad \text{PCT}' = \langle \text{P}', \text{LS}' \rangle$$

$$(6.4.6a.ii) \quad \text{PCT}'' = \langle \text{P}', \text{LS}'' \rangle$$

$$(6.4.6a.iii) \quad \text{LS}'' = \text{LS}' \cup \{\text{PP}''\}$$

$$(6.4.6a.iv) \quad \text{PP}'' \leq \text{PP}$$

$$(6.4.6a.v) \quad \text{same side conditions as in (4.4.16a.i) through (4.4.16a.vi).}$$

$$(6.4.6b) \quad \text{same rule as in (6.4.6a).}$$

$$(6.4.6b.i) \quad \text{where} \quad \text{PCT}' = \langle \text{P}', \text{LS}' \rangle$$

$$(6.4.6b.ii) \quad \text{PCT}'' = \langle \text{P}'', \text{LS}' \rangle$$

$$(6.4.6b.iii) \quad \text{PP}'' > \text{PP}$$

$$(6.4.6b.iv) \quad \text{P}'' = \text{P}' \sqcap \{\text{PP}'' \mapsto \text{FT}_{\text{PP}''}^{\text{PP}}\}$$

$$(6.4.6b.v) \quad \text{same side-conditions as in (4.4.16a.i) through (4.4.16a.vi).}$$

Observation 6.4.7 (Exception attribute certification). Because the verification of the exception attribute performs a type check which is independent of the code, it does not contribute to the accumulated jump set, and as such collapses with the already specified attribute verification in Definition 4.4.20.

Remark 6.4.8 (Error certification). These can be certified as any other group of unchecked exception, though under the same constraints which are already true for exception verification as discussed in Remark 4.4.22.

We proceed by proving some equivalence properties between exception standard verification rules of (BV) and exception certification rules of (LBC), and then, between exception certification rules of (LBC) and exception lightweight verification rules of (LBV).

Observation 6.4.9 (Side condition invariant). We notice, that Observation 6.3.12 also apply for exception rules. Thus, we consider symbols, functions, and abbreviations to have the same meaning in the two exception systems.

Proposition 6.4.10. The (LBC) and (BV) establish the same safety guarantees at the same program points on the same set of raised exceptions.

Verifying rule	Certifying rule	Catch	Type constraint
(4.4.12a)	(6.4.4a)	no	—
(4.4.12b)	(6.4.4b)	no	—
(4.4.12c)	(6.4.4c)	no	—
(4.4.12d)	(6.4.4d)	no	—
(4.4.12e)	(6.4.4e)	no	—
(4.4.14a)	(6.4.5a), (6.4.5b)	definite	$CH \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$
(4.4.16a)	(6.4.6a), (6.4.6b)	potential	$CH \vdash \text{FTA}(\text{PP}'') \sqsubseteq \text{FT}_{\text{PP}''}^{\text{PP}}$

Figure 6.6: Exception rule type safety.

Rule	Handling at PP''	Label update	Pending update
(6.4.4a)	—	—	—
(6.4.4b)	—	—	—
(6.4.4c)	—	—	—
(6.4.4d)	—	—	—
(6.4.4e)	—	—	—
(6.4.5a)	backwards	$LS \cup \{\text{PP}''\}$	—
(6.4.5b)	forwards	—	$P \sqcap \{\text{PP}'' \mapsto \text{FT}_{\text{PP}''}^{\text{PP}}\}$
(6.4.6a)	backwards	$LS \cup \{\text{PP}''\}$	—
(6.4.6b)	forwards	—	$P \sqcap \{\text{PP}'' \mapsto \text{FT}_{\text{PP}''}^{\text{PP}}\}$

Figure 6.7: Exception certification updates.

Proof. Easily seen from Figure 6.6 by the exhaustive listing of (BV) and (LBC) rules, the corresponding catch situations they cover, and the type safety constraints they introduce. \square

Proposition 6.4.11. For any program points PP and PP'' of some verifiable method, both (LBC) and (LBV) update P with the same frame type in the same program points.

Proof. From the summaries in Figures 6.7 6.8, 6.9, and 6.10, we observe that P is updated by the same frame type, and by forward jump exception-rules which correspond one-to-one between the two verification systems (that is whenever PP'' is a forward handle-position in relation to PP , where the exception is launched). \square

Rule	Handling at PP''	Saved Constraint	Pending update
(5.4.5a)	—	—	—
(5.4.5b)	—	—	—
(5.4.5c)	—	—	—
(5.4.5d)	—	—	—
(5.4.5e)	—	—	—
(5.4.6a)	backwards	$CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$	—
(5.4.6b)	forwards	—	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$
(5.4.7a)	backwards	$CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$	—
(5.4.7b)	forwards	—	$P \sqcap \{PP'' \mapsto FT_{PP''}^{PP}\}$

Figure 6.8: Exception lightweight updates and type safety.

In accordance with Proposition 6.3.33 we finally show an important property.

Proposition 6.4.12. For any program points PP and PP'' of some verifiable method, there is a one-to-one correspondence between the exception rules which check the saved constraint in (LBV), that is $CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$, and the exception rules which update the backward label set in (LBC), that is $LS \cup \{PP''\}$

Proof. From the summaries in Figures 6.7, 6.8, 6.9, and 6.10, we observe that LS is extended by the same program point PP'' by the exception rules in (LBC) whenever the constraint $CH \vdash S(PP'') \sqsubseteq FT_{PP''}^{PP}$ is checked in (LBV) by backward jump exception-rules which correspond one-to-one between the two verification systems (that is whenever PP'' is a backward handle-position in relation to PP , where the exception is launched). \square

6.5 The Example

Let us assume the verification context, frame type assignment, class hierarchy, and other class file formalization details as listed in Figure 4.6 on page 91. We can now formally show that the `cksum()` method certifies with the certificate already listed in Proposition 5.5.1.

Certifying rule	(6.4.4a)	(6.4.4b)	(6.4.4c)	(6.4.4d)	(6.4.4e)
Lightweight rule	(5.4.5a)	(5.4.5b)	(5.4.5c)	(5.4.5d)	(5.4.5e)

Figure 6.9: No-catch exception-rule correspondence.

Catch situation	Definite	Definite	Potential	Potential
Certifying rule	(6.4.5a)	(6.4.5b)	(6.4.6a)	(6.4.6b)
Lightweight rule	(5.4.6a)	(5.4.6b)	(5.4.7a)	(5.4.7b)

Figure 6.10: Catch exception-rule correspondance.

Proposition 6.5.1 (cksum lightweight certifies). The $\text{cksum}()$ method M^{ck} lightweight certifies in the context Γ^{ck} (both defined in Figure 4.6) with the frame type assignment FTA^{ck} (defined in Figure 4.5) and certificate CE^{ck} (defined in Proposition 5.5.1). Formally: $\Gamma^{\text{ck}} \vdash_{\text{lbv}} M^{\text{ck}}, \text{FTA}^{\text{ck}}, \text{CE}^{\text{ck}}$.

Proof. Assume Γ^{ck} , M^{ck} , FTA^{ck} , and CE^{ck} as in the Proposition. Then the lightweight rules of this chapter can be used to construct the proof

$$(6.5.2) \quad \frac{\frac{\boxed{\text{A.3.1}}}{\Omega_{\text{light}}^{\text{ck}} \vdash \langle 0, \text{FT}_0^{\text{ck}} \rangle, C^{\text{ck}}, \emptyset, \langle P_0^{\text{ck}}, S_0^{\text{ck}} \rangle \xRightarrow{\text{Itsafe}} \text{PPS}^{\text{ck}}, \langle P_0^{\text{ck}}, S_{20}^{\text{ck}} \rangle} (6.2.9c)}{\Gamma^{\text{ck}} \vdash M^{\text{ck}}, \text{CE}^{\text{ck}}} (6.2.4a)$$

where $\Omega_{\text{light}}^{\text{ck}}$, Δ^{ck} , P_0^{ck} , S_0^{ck} , and S_{20}^{ck} from Proposition 5.5.1, and where $\boxed{\text{A.3.1}}$ denotes the full proof tree given in Appendix A.3. \square

Chapter 7

The Prototype Implementation

In this chapter, we have implemented a Java prototype for our lightweight verifier from Chapter 5. We notice, that the implementation is original to the thesis.

In Section 7.1, we begin with a brief description of our implementation strategy. Then, in Section 7.2, we present a definition of the Java interfaces which we use to implement the class file description sorts from Chapter 3. In Section 7.3, we subsequently show the prototype implementation of our semantic lightweight verification rules from Chapter 5, and in Section 7.4, follows the main program and “glue” code to access the standard Java class files. Finally, in Section 7.5, we present a user’s manual for the program.

7.1 Implementation Strategy

In order to investigate the realizability of the lightweight verification technique on the execution platform, we notice that the lightweight checker formalization is *tail recursive* in the code sequence, thus can be effectively implemented by a `while` loop.

Observation 7.1.1 (The lightweight verifier space bound). We have that a compiled method, with the frame type constraints ML and MS , and with the number of backward jumps $\#LS$, and forward jumps q , verifies in (almost) linear time using space bounded by

$$(7.1.1a) \quad (ML + MS) \times (1 + \#LS + q)$$

7.2 The Class File Context

The section contains the source code for the Java interfaces which define how the underlying class access library must implement the basic sorts of Chapter 3. (All of these interfaces are implemented using `BCEL.java` of 7.4.3 to access standard class files but they could be implemented by any class file context implementation.)

Remark 7.2.1 (Primitive types). Figure 7.1 shows the sorts are implemented by basic Java types. Notice that the sorts used by `Type` use *internalized* Java `Strings` to permit the use of `==` to test equality (see Section 7.3.2 below for the details).

Sort	definition	Java type
OpCode	(3.1.4b)	int
ClassIdent	(3.2.2a)	String (internalized)
Identifier	(3.2.2k)	String
MaxStack	(3.2.6e)	int
MaxLocals	(3.2.6f)	int
PPoint	(3.2.8d)	int
Type	(3.2.2f)	String (internalized)
ReturnType	(3.2.2f)	String (internalized)

Figure 7.1: Sorts implemented by basic Java types.

Source 7.2.2 (StdContext.java). Defines interface for implementations of the StdContext and ClassFileContext sorts of Definition 3.2.1.

```

1 interface StdContext {
2
3     // Class constant pool (cp).
4     ConstPool getConstPool();
5
6     // Class name (cid).
7     String getClassIdent();
8
9     // Class hierarchy (ch).
10    ClassHier getClassHier();
11 }

```

Source 7.2.3 (ConstPool.java). Defines interface for implementations of the ConstPool and Item sorts of Definition 3.2.2.

```

1 interface ConstPool {
2
3     // Length of constant pool (max(Dom(cp))).
4     int getConstPoolLength();
5
6     // Get class identifier at index.
7     // Throws VerifError if item is not a class identifier.
8     String getClassIdent(int index) throws VerifError;
9
10    // Get field reference at index.
11    // Throws VerifError if item is not a field reference.
12    FieldRef getFieldRef(int index) throws VerifError;
13
14    // Get method reference at index.
15    // Throws VerifError if item is not a method reference.
16    MethRef getMethRef(int index) throws VerifError;
17

```

```

18 // Get integer at index.
19 // Throws VerifError if item is not an integer.
20 int getInteger(int index) throws VerifError;
21
22 // Get type at index.
23 // Throws VerifError if item is not a type.
24 String getType(int index) throws VerifError;
25
26 // Item in string form (for diagnostics).
27 String item(int index); // item as string
28 }

```

Source 7.2.4 (FieldRef.java). Defines interface for implementations of the FieldRef sort of (3.2.2b).

```

1 interface FieldRef {
2
3 // Class containing field.
4 String getClassIdent();
5
6 // Field name.
7 String getIdent();
8
9 // Field type.
10 String getType();
11 }

```

Source 7.2.5 (MethRef.java). Defines interface for implementations of the MethRef sort of (3.2.2c).

```

1 interface MethRef extends MethSig {
2
3 // Class containing method.
4 String getClassIdent();
5
6 // Method name.
7 String getIdent();
8
9 // Method return type.
10 String getReturnType();
11 }

```

Source 7.2.6 (MethSig.java). Defines interface for implementations of the MethSig sort of (3.2.2d).

```

1 // Implements
2 interface MethSig {
3
4 // Name of method (id).
5 String getIdent();
6

```

```

7    // Number of arguments (k).
8    int getArgumentCount();
9
10   // Argument type (ti).
11   String getArgumentType(int i);
12 }

```

Source 7.2.7 (Method.java). Defines interface for implementations of the Method sort of (3.2.6a).

```

1 interface Method {
2
3    // Method signature (msig).
4    MethSig getMethSig();
5
6    // Method return type (rt).
7    String getReturnType();
8
9    // Method code attribute (ca).
10   CodeAtt getCodeAtt();
11
12   // Method exception attribute (ea).
13   ExcAtt getExcAtt();
14 }

```

Source 7.2.8 (ExcAtt.java). Defines interface for implementations of the ExcAtt sort of (3.2.6b).

```

1 interface ExcAtt {
2
3    // Throws VerifError if excep is not in the exception attribute.
4    void checkContains(String excep) throws VerifError;
5 }

```

Source 7.2.9 (CodeAtt.java). Defines interface for implementations of the CodeAtt and MaxFrame sorts of (3.2.6c) and 3.2.6d.

```

1 interface CodeAtt {
2
3    // Size of stack [sic].
4    int getMaxStack();
5
6    // Number of local variables.
7    int getMaxLocals();
8
9    // Bytecode.
10   Code getCode();
11
12   // Table of exception handlers.
13   ExcTable getExcTable();
14 }

```

Source 7.2.10 (Code.java). Defines interface for implementations of the Code and CodeSeq sorts of Definition 3.2.8 except that the instructions of the Code are addressed by program point rather than kept in a list.

```

1 interface Code {
2
3     // Number of last byte in Code.
4     int getMaxCode();
5
6     // Opcode for bytecode instruction starting at pp.
7     int getOpcode(int pp);
8
9     // Length (in bytes) of bytecode instruction starting at pp.
10    int getLength(int pp);
11
12    // Return index or jump offset argument value of bytecode instruction
13    // starting at pp.
14    int getArgument(int pp);
15
16    // Printable representation of instruction starting at pp.
17    String instruction(int pp);
18 }

```

Source 7.2.11 (ExcTable.java). Defines interface for implementations of the ExcTable and related sorts of Definition 3.2.12.

```

1 interface ExcTable {
2
3     // Number of exception handlers.
4     int getExcCount();
5
6     // Whether pp is within the range handled by handler number n.
7     boolean inTryRange(int n, int pp);
8
9     // Type handled by handler number n.
10    String getCatchType(int n) throws VerifError;
11
12    // Start of handler number n.
13    int getCatchPP(int n);
14 }

```

Source 7.2.12 (ClassHier.java). Defines interface for implementations of the ClassHier sort and related ordering relation \leq_{CH} of Section 3.3.

```

1 interface ClassHier {
2
3     // Whether a type is a (non-array) class in the class hierarchy.
4     boolean isClass(String type);

```

```

5
6 // Whether class is superclass of another (implements >=:).
7 // Throws exception if isClass fails on either parameter.
8 boolean compatible(String cid, String sub_cid) throws VerifError;
9
10 // Largest class with both cid1 and cid2 as subclasses.
11 // Throws exception if isClass fails on either parameter.
12 String meet(String cid1, String cid2) throws VerifError;
13
14 // Output.
15 void printClassHier(String prefix);
16 }

```

7.3 Lightweight Bytecode Verification

This section contains source of the Java classes used to implement the sorts of Chapters 5.

Remark 7.3.1 (Failure). Failure is indicated in one of two ways: either by a false return value and/or by throwing a `VerifError` exception (defined in Section 7.3.11): the latter is used as a shortcut in cases where failure is known to imply failure of the entire verification.

Source 7.3.2 (Type.java). Implements Type assignment compatibility ordering \sqsubseteq following Definition 4.1.10.

```

1 class Type {
2
3 // Constant class names.
4 final static String object = "Ljava/lang/Object;";
5 final static String objectArray = "[Ljava/lang/Object;";
6
7 final static String throwable = "Ljava/lang/Throwable;";
8 final static String exception = "Ljava/lang/Exception;";
9 final static String runtime = "Ljava/lang/RuntimeIOException;";
10 final static String nullPointerException = "Ljava/lang/NullPointerException;";
11 final static String arrayIndexOutOfBoundsException =
12     "Ljava/lang/ArrayIndexOutOfBoundsException;";
13 final static String arrayStoreException = "Ljava/lang/ArrayStoreException;";
14 final static String negativeArraySizeException =
15     "Ljava/lang/NegativeArraySizeException;";
16 final static String classCastException = "Ljava/lang/ClassCastException;";
17
18 // Constant types.
19 final static String integer = "I";
20 final static String integerArray = "[I";
21
22 // Abstract types.
23 final static String nul = "null";

```

```
24 final static String top = "top";
25 final static String bot = "bot";
26 final static String botArray = "[bot]";
27
28 // Void pseudo-types.
29 final static String voi = "V";
30
31 // Return type, i.e., type-internalized String, with name typeName.
32 // (All the following methods require (and return) such types.)
33 static String type(String typeName) {
34     return typeName.intern();
35 }
36
37 // Whether type is the bot Type.
38 static boolean isBot(String type) {
39     return (type == bot);
40 }
41
42 // Whether type is the null Type.
43 static boolean isNull(String type) {
44     return (type == nul);
45 }
46
47 // Whether type is the void Type.
48 static boolean isVoid(String type) {
49     return (type == voi);
50 }
51
52 // Whether type is an integer.
53 static boolean isInteger(String type) {
54     return (type == integer);
55 }
56
57 // Whether type is the Object class.
58 static boolean isObject(String type) {
59     return (type == object);
60 }
61
62 // Whether type is some array.
63 static boolean isArray(String type) {
64     return (type != null && type.length() > 0 && type.charAt(0) == '[');
65 }
66
67 // Whether type is an array of integers.
68 static boolean isIntegerArray(String type) {
69     return (type == integerArray);
70 }
```

```

71
72 // Whether type is an array of Object references.
73 static boolean isArray(String type) {
74     return (type == objectArray);
75 }
76
77 // Whether type is an array of reference Types to some class in ch.
78 static boolean isArray(ClassHier ch, String type) throws VerifError {
79     try {
80         return (isArray(type) && ch.isClass(arrayBase(type)));
81     }
82     catch (VerifError x) {
83         return false; // cannot happen
84     }
85 }
86
87 // Whether type is the top Type.
88 static boolean isTop(String type) {
89     return (type == top);
90 }
91
92 // Base Type of arrayType. Throws VerifError if not isArray(arrayType).
93 static String arrayBase(String arrayType) throws VerifError {
94     if (arrayType == botArray) return bot;
95     if (arrayType == objectArray) return object;
96     if (arrayType == integerArray) return integer;
97     return type(arrayType.substring(1)); // expensive hack: do better!
98 }
99
100 // Array Type with elements of type.
101 static String arrayOf(String type) {
102     return type("[" + type);
103 }
104
105 // Whether t is a supertype of sub_t given the class hierarchy ch.
106 static boolean compatible(ClassHier ch, String t, String sub_t) {
107     try {
108         if (t == sub_t || t == bot || sub_t == top) return true;
109         if (ch.isClass(t) && ch.isClass(sub_t))
110             return ch.compatible(t, sub_t);
111         if (t == object && isArray(sub_t))
112             return true;
113         if (isArray(t) && isArray(sub_t))
114             return compatible(ch, arrayBase(t), arrayBase(sub_t));
115     }
116     catch (VerifError e) {} // cannot happen
117     return false;

```

```

118     }
119
120     // Largest type that is smaller than both t1 and t2.
121     static String meet(ClassHier ch, String t1, String t2) {
122         String t = bot;
123         try {
124             if (t1 == t2)
125                 t = t1;
126             else if (t1 == top)
127                 t = t2;
128             else if (t2 == top)
129                 t = t1;
130             else if (t1 == object && isArray(t2))
131                 t = t1;
132             else if (isArray(t1) && t2 == object)
133                 t = t2;
134             else if (isArray(t1) && isArray(t2))
135                 t = arrayOf(meet(ch, arrayBase(t1), arrayBase(t2)));
136             else if (ch.isClass(t1) && ch.isClass(t2))
137                 t = ch.meet(t1, t2);
138         }
139         catch (VerifError e) {} // cannot happen
140         //System.out.println("meet("+t1+", "+t2+") = "+t);
141         return t;
142     }
143
144 }

```

Source 7.3.3 (FrameType.java). Implements FrameType, StackType, and LocalType sorts, and the extension of \sqsubseteq to frame types, following Definitions 4.1.10 through 4.2.3.

```

1  class FrameType {
2
3     // STATE.
4
5     ClassHier ch;
6
7     boolean isTop;
8     boolean isBot;
9     int stackLength;
10    String[] stack;
11    String[] locals;
12
13    // CONSTRUCTORS.
14
15    // Allocate uninitialized FrameType.
16    FrameType(StdContext gamma, int ms, int ml) throws VerifError {
17        ch = gamma.getClassHier();

```

```

18     stack = new String[ms];
19     locals = new String[ml];
20 }
21
22 // Allocate FrameType with copy of another.
23 FrameType(FrameType ft) throws VerifError {
24     ch = ft.ch;
25     stack = new String[ft.stack.length];
26     locals = new String[ft.locals.length];
27     set(ft);
28 }
29
30 // INITIALIZATION METHODS.
31
32 // Set to the copy of another FrameType.
33 public void set(FrameType ft) {
34     isTop = ft.isTop;
35     isBot = ft.isBot;
36     stackLength = ft.stackLength;
37     for (int i = 0; i < stack.length; ++i) stack[i] = ft.stack[i];
38     for (int i = 0; i < locals.length; ++i) locals[i] = ft.locals[i];
39 }
40
41 // Set to the top frame type.
42 public void setTop() {
43     isTop = true;
44     isBot = false;
45     stackLength = 0;
46     String bot = Type.bot;
47     for (int i = 0; i < stack.length; ++i) stack[i] = bot;
48     for (int i = 0; i < locals.length; ++i) locals[i] = bot;
49 }
50
51 // Set to the bottom FrameType.
52 public void setBot() {
53     isTop = false;
54     isBot = true;
55     stackLength = 0;
56     String bot = Type.bot;
57     for (int i = 0; i < stack.length; ++i) stack[i] = bot;
58     for (int i = 0; i < locals.length; ++i) locals[i] = bot;
59 }
60
61 // Set to the initial frame type corresponding to the class and method signature.
62 public void init(String cid, MethSig msig) throws VerifError {
63     isTop = false;
64     isBot = false;

```

```

65     stackLength = 0;
66
67     String bot = Type.bot;
68     for (int i = 0; i < stack.length; ++i) stack[i] = bot;
69
70     int argumentCount = msig.getArgumentCount();
71     if (argumentCount >= locals.length)
72         throw VerifError.getInstance("cannot fit arguments in frame local variables");
73     locals[0] = cid;
74     for (int i = 1; i <= argumentCount; ++i)
75         locals[i] = msig.getArgumentType(i-1);
76     for (int i = argumentCount+1; i < locals.length; ++i)
77         locals[i] = bot;
78 }
79
80 // HIGH-LEVEL MUTATION METHODS.
81
82 // Update FrameType to be less than the other FrameType.
83 public void meetWith(FrameType other) {
84     //System.out.println(" "+stringFrameType()+" meet "+other.stringFrameType()+" :");
85
86     // Frame type unchanged if bottom or other is top.
87     if (isBot || other.isTop) {
88         //System.out.println(" = "+stringFrameType());
89         return;
90     }
91
92     // FrameType forced to bottom if other is bottom or the stack/locals lengths differ.
93     if (other.isBot || stack.length != other.stack.length
94         || locals.length != other.locals.length)
95     {
96         setBot();
97         //System.out.println(" = "+stringFrameType());
98         return;
99     }
100
101     // Frame type copied over if only this is top.
102     if (isTop) {
103         set(other);
104         //System.out.println(" = "+stringFrameType());
105         return;
106     }
107
108     // Otherwise build pointwise meet.
109     for (int i = 0; i < stackLength; ++i)
110         stack[i] = Type.meet(ch, stack[i], other.stack[i]);
111     for (int i = 0; i < locals.length; ++i)

```

```

112     locals[i] = Type.meet(ch, locals[i], other.locals[i]);
113     //System.out.println(" = " +stringFrameType());
114 }
115
116 // BASIC ACCESS AND MUTATION METHODS.
117
118 // Mutate frame type to have one more type on the stack.
119 // Throws VerifError if this is impossible.
120 void stackPush(String type) throws VerifError {
121     if (isBot || isTop)
122         throw VerifError.getInstance("cannot_push_onto_undefined_or_wrong_stack");
123     if (stackLength == stack.length)
124         throw VerifError.getInstance("cannot_push_further_onto_full_stack");
125     stack[stackLength++] = type;
126 }
127
128 // Top stack type. Throws VerifError if stack is empty.
129 String stackTop() throws VerifError {
130     if (isBot || isTop)
131         throw VerifError.getInstance("undefined_or_wrong_stack_have_no_top_element");
132     if (stackLength == 0)
133         throw VerifError.getInstance("no_top_element_on_empty_stack");
134     return stack[stackLength - 1];
135 }
136
137 // Mutate frame type to have one less type on the stack.
138 // Return the pop'd type.
139 // Throws VerifError if this is impossible.
140 String stackPop() throws VerifError {
141     if (isBot || isTop)
142         throw VerifError.getInstance("cannot_pop_from_undefined_or_wrong_stack");
143     if (stackLength == 0)
144         throw VerifError.getInstance("cannot_pop_from_empty_stack");
145     String type = stack[--stackLength];
146     stack[stackLength] = Type.bot;
147     return type;
148 }
149
150 // Set stack to contain just a single type.
151 void stackSet(String type) throws VerifError {
152     if (isBot || isTop)
153         throw VerifError.getInstance("cannot_set_local_variable_of_undefined_or_wrong_frame");
154     stack[0] = type;
155     stackLength = 1;
156 }
157
158 // Set local variable n's type. Throws VerifError if no such local variable.

```

```

159 void localsSet(int n, String type) throws VerifError {
160     if (isBot || isTop)
161         throw VerifError.getInstance("cannot_set_local_variable_of_undefined_or_wrong_frame");
162     if (n < 0 || n >= locals.length)
163         throw VerifError.getInstance("local_variable_number_out_of_bounds");
164     locals[n] = type;
165 }
166
167 // Local variable n's type. Throws VerifError if no such local variable.
168 String localsGet(int n) throws VerifError {
169     if (isBot || isTop)
170         throw VerifError.getInstance("cannot_get_local_variable_of_undefined_or_wrong_frame");
171     if (n < 0 || n >= locals.length)
172         throw VerifError.getInstance("local_variable_number_out_of_bounds");
173     return locals[n];
174 }
175
176 // CHECK FILTERS.
177
178 // Return first type argument.
179 // Throws VerifError if the first type is not the same as the second.
180 String checkSame(String type, String type2) throws VerifError {
181     if (type != type2)
182         throw VerifError.getInstance(type + "is_not_the_same_as_" + type2);
183     return type;
184 }
185
186 // Return first type argument.
187 // Throws VerifError if the first type is not a supertype of the second.
188 String checkSuper(String type, String sub_type) throws VerifError {
189     if (!Type.compatible(ch, type, sub_type))
190         throw VerifError.getInstance(type + "is_not_supertype_of_" + sub_type);
191     return type;
192 }
193
194 // Return type argument.
195 // Throws VerifError if the first type is not a subtype of the second.
196 String checkSub(String type, String super_type) throws VerifError {
197     if (!Type.compatible(ch, super_type, type))
198         throw VerifError.getInstance(type + "is_not_subtype_of_" + super_type);
199     return type;
200 }
201
202 // String of frame type.
203 String stringFrameType() {
204     StringBuffer sb = new StringBuffer();
205     if (isBot)

```

```

206     sb.append("bot");
207     else if (isTop)
208         sb.append("top");
209     else {
210         sb.append("<");
211         if (stackLength == 0)
212             sb.append("*");
213         else
214             for (int i = 0; i < stackLength; ++i)
215                 sb.append((i == 0 ? "" : ".") + stack[i]);
216         sb.append(",");
217         if (locals.length == 0)
218             sb.append("*");
219         else
220             for (int i = 0; i < locals.length; ++i)
221                 sb.append((i == 0 ? "" : ".") + locals[i]);
222         sb.append(">");
223     }
224     return sb.toString();
225 }
226
227 }
```

Source 7.3.4 (FrameTypeMap.java). Implements FrameTypeMap which provides the basis for the Pending and Saved sorts, *cf.* Definition 5.1.10.

```

1 class FrameTypeMap {
2
3     // State.
4
5     private StdContext gamma;
6
7     private int[] pps;
8     private FrameType[] fts;
9
10    // Constructor.
11
12    // Construct FrameTypeMap with a maximal domain size.
13    FrameTypeMap(StdContext gamma, int size) {
14        this.gamma = gamma;
15        pps = new int[size];
16        for (int i = 0; i < size; ++i) pps[i] = -1;
17        fts = new FrameType[size];
18    }
19
20    // Methods.
21
22    // Set the mapping for pp to ft.
```

```

23 // Throws VerifError if there are no more slots.
24 void set(int pp, FrameType ft) throws VerifError {
25     for (int i = 0; i < pps.length; ++i)
26         if (pps[i] == pp) {
27             fts[i].meetWith(ft);
28             return;
29         }
30     for (int i = 0; i < pps.length; ++i)
31         if (pps[i] == -1) {
32             pps[i] = pp;
33             fts[i] = new FrameType(ft);
34             fts[i].set(ft);
35             return;
36         }
37     throw VerifError.getInstance("no more FrameType map slots");
38 }
39
40 // Whether there are no mappings.
41 boolean isEmpty() {
42     for (int i = 0; i < pps.length; ++i)
43         if (pps[i] != -1) return false;
44     return true;
45 }
46
47 // Check if there is a mapping for pp.
48 boolean contains(int pp) {
49     for (int i = 0; i < pps.length; ++i)
50         if (pps[i] == pp) return true;
51     return false;
52 }
53
54 // Look up the mapping of pp.
55 // Throws VerifError if map undefined for pp.
56 FrameType get(int pp) throws VerifError {
57     for (int i = 0; i < pps.length; ++i)
58         if (pps[i] == pp) return fts[i];
59     throw VerifError.getInstance("map undefined for " + pp);
60 }
61
62 // Remove mapping of pp (including the FrameType it mapped to).
63 // Throws VerifError if map undefined for pp.
64 void remove(int pp) throws VerifError {
65     for (int i = 0; i < pps.length; ++i)
66         if (pps[i] == pp) {
67             pps[i] = -1;
68             return;
69         }

```

```

70     throw VerifError.getInstance("map_undefined_for_" + pp);
71 }
72
73 // Print frame type map.
74 String frameTypeMap() {
75     if (pps.length == 0)
76         return("{}");
77     else {
78         StringBuffer sb = new StringBuffer();
79         for (int i = 0; i < pps.length; ++i)
80             if (pps[i] >= 0)
81                 sb.append((i==0 ? "" : ", ")
82                     + pps[i] + ":"
83                     + fts[i].stringFrameType()
84                     + (i+1==pps.length?"}":""));
85         return sb.toString();
86     }
87 }
88
89 }

```

Source 7.3.5 (Pending.java). Implements Saved sort (5.1.10c).

```

1 class Pending extends FrameTypeMap {
2
3     // Construct Pending object for forward jumps cert.
4     Pending(StdContext gamma, Cert cert, int pendingSize) {
5         super(gamma, pendingSize);
6     }
7 }

```

Source 7.3.6 (Saved.java). Implements Pending sort (5.1.10b).

```

1 class Saved extends FrameTypeMap {
2
3     // Construct holder of saved frame types for backwards jumps labeled by cert.
4     Saved(StdContext gamma, Cert cert) {
5         super(gamma, cert.labelCount());
6     }
7 }

```

Source 7.3.7 (LightContext.java). Interface to context of lightweight method verification of Definition 5.2.3 (implemented by 7.4.3 below).

```

1 class LightContext implements MethContext {
2
3     // Fields.
4     StdContext gamma;

```

```
5   String rt; // Return Type
6   int ms; // Max Stack
7   int ml; // Max Locals
8   ExcAtt ea;
9   ExcTable et;
10  Cert ce;
11
12  // Static instantiator...
13  private LightContext() {}
14  private static LightContext self = new LightContext();
15  static LightContext make(StdContext gamma, Method m, Cert ce) {
16      self.gamma = gamma;
17      self.rt = m.getReturnType();
18      CodeAtt ca = m.getCodeAtt();
19      self.ms = ca.getMaxStack();
20      self.ml = ca.getMaxLocals();
21      self.ea = m.getExcAtt();
22      self.et = ca.getExcTable();
23      self.ce = ce;
24      return self;
25  }
26
27  // Generic class context.
28  StdContext getStdContext() {
29      return gamma;
30  }
31
32  // Method context.
33  MethContext getMethContext() {
34      return this;
35  }
36
37  // Certificate.
38  Cert getCert() {
39      return ce;
40  }
41
42  // MethContext implementation.
43
44  public String getReturnType() {
45      return rt;
46  }
47
48  public int getMaxStack() {
49      return ms;
50  }
51
```

```

52     public int getMaxLocals() {
53         return ml;
54     }
55
56     public ExcAtt getExcAtt() {
57         return ea;
58     }
59
60     public ExcTable getExcTable() {
61         return et;
62     }
63
64 }

```

Source 7.3.8 (Cert.java). Interface to lightweight certificate Cert sort and the related FrameTypeCert, Labels, and Label sorts of Definition 5.1.7.

```

1  interface Cert {
2
3      // Number of labels?
4      int labelCount();
5
6      // Does the certificate label program point pp?
7      boolean isLabel(int pp);
8
9      // Modify ft to be equal to the smaller frame type that the certificate associates
10     // to the pp program point. Throws VerifError if the modification would
11     // not yield a smaller frame type.
12     void applyDeltaTo(FrameType ft, int pp) throws VerifError;
13
14     // Pending count.
15     int getPendingSize();
16
17     // Print certificate (for debugging).
18     void printCert(String prefix);
19 }

```

Source 7.3.9 (MethLbv.java). Implements lightweight bytecode verification for a method of Definition 5.2.5. Uses a while loop for the tail recursive inference rules as discussed in Remark 5.2.9.

```

1  class MethLbv {
2
3      // Verify method.
4      static void verify(StdContext gamma, Method m, Cert ce, boolean verbose) throws VerifError
5      {
6          LightContext omega = LightContext.make(gamma, m, ce);
7          MethContext delta = omega.getMethContext();
8

```

```

 9      // Allocate current FrameType and initialize to initial value.
10      FrameType ft = new FrameType(gamma, delta.getMaxStack(), delta.getMaxLocals());
11      ft.init(gamma.getClassIdent(), m.getMethSig());
12
13      // Access Code properties.
14      CodeAtt ca = m.getCodeAtt();
15      Code c = ca.getCode();
16      int pp_end = c.getMaxCode();
17
18      // Allocate saved and pending sets.
19      Saved S = new Saved(gamma, ce);
20      Pending P = new Pending(gamma, ce, ce.getPendingSize());
21
22      // Verify all instructions.
23      for (int pp = 0; pp <= pp_end; pp += c.getLength(pp)) {
24          ce.applyDeltaTo(ft, pp);
25          if (ce.isLabel(pp) || P.contains(pp)) {
26              if (P.contains(pp)) {
27                  ft.meetWith(P.get(pp));
28                  P.remove(pp);
29              }
30              if (ce.isLabel(pp))
31                  S.set(pp, ft);
32          }
33
34          if (verbose) {
35              System.out.println("└─" + ft.stringFrameType());
36              System.out.println(pp + "┆:" + c.instruction(pp));
37          }
38
39          InsLbv.verify(omega, pp, ft, c, S, P);
40          //P.printFrameTypeMap(" P =");
41          //S.printFrameTypeMap(" S =");
42      }
43      if (verbose)
44          System.out.println("└─" + ft.stringFrameType());
45
46      // Final consistency checks.
47      if (!P.isEmpty())
48          throw VerifError.getInstance("some pending constraints not resolved");
49  }
50 }

```

Source 7.3.10 (MethContext.java). Defines interface for MethContext sort of Definition 4.2.6.

```

1  interface MethContext {
2
3      // Method return type (rt).

```

```

4   String getReturnType();
5
6   // Method max stack length (ms).
7   int getMaxStack();
8
9   // Method max local variable length (ml).
10  int getMaxLocals();
11
12  // Method exception attribute (ea).
13  ExcAtt getExcAtt();
14
15  // Method exception table (et).
16  ExcTable getExcTable();
17 }

```

Source 7.3.11 (VerifError.java). Exception class used for lightweight bytecode verification failure.

```

1  class VerifError extends Exception {
2
3      // Factory (permits singleton).
4      static VerifError getInstance(String m) {
5          return new VerifError(m);
6      }
7
8      // Constructor.
9      private VerifError(String m) {
10         super(m);
11     }
12
13 }

```

Source 7.3.12 (InsLbv.java). Implements lightweight bytecode verification of a single instruction as specified in Section 5.3.

```

1  class InsLbv {
2
3      // Verify instruction. Mutates ft, S, and P.
4      // Throws VerifError on failure.
5      static void verify(LightContext omega,
6                        int pp, FrameType ft, Code c, Saved S, Pending P)
7          throws VerifError
8      {
9          StdContext gamma = omega.getStdContext();
10         ClassHier ch = gamma.getClassHier();
11         ConstPool cp = gamma.getConstPool();
12
13         MethContext delta = omega.getMethContext();

```

```
14     ExcTable et = delta.getExcTable();
15     ExcAtt ea = delta.getExcAtt();
16
17     Cert ce = omega.getCert();
18
19     switch (c.getOpcode(pp)) {
20
21         // STACK INSTRUCTIONS.
22
23         case 1: // aconst_null
24             ft.stackPush(Type.nul);
25             break;
26
27         case 3: // iconst_0
28         case 4: // iconst_1
29             ft.stackPush(Type.integer);
30             break;
31
32         case 87: // pop
33             ft.stackPop();
34             break;
35
36         case 89: // dup
37             ft.stackPush(ft.stackTop());
38             break;
39
40         case 96: // iadd
41         case 100: // isub
42             ft.checkSame(ft.stackPop(), Type.integer);
43             ft.checkSame(ft.stackTop(), Type.integer); // left as type of result
44             break;
45
46         // LOCAL VARIABLE INSTRUCTIONS.
47
48         case 21: // iload
49             ft.stackPush(ft.checkSame(ft.localsGet(c.getArgument(pp)), Type.integer));
50             break;
51
52         case 25: // aload
53             ft.stackPush(ft.checkSub(ft.localsGet(c.getArgument(pp)), Type.object));
54             break;
55
56         case 54: // istore
57             ft.localsSet(c.getArgument(pp), ft.checkSame(ft.stackPop(), Type.integer));
58             break;
59
60         case 58: // astore
```

```

61     ft.localsSet(c.getArgument(pp), ft.checkSub(ft.stackPop(), Type.object));
62     break;
63
64     // ARRAY HEAP/STACK INSTRUCTION.
65
66     case 188: // newarray
67         checkThrow(gamma, delta, ft, false, pp, Type.negativeArraySizeException, S, P);
68         if (c.getArgument(pp) != 10) // we only support array of integer (T_INT)
69             throw VerifError.getInstance("newarray_on_non-integer_base_type_" + c.getArgument(pp));
70         ft.checkSame(ft.stackPop(), Type.integer);
71         ft.stackPush(Type.integerArray);
72         break;
73
74     case 189: // anewarray
75         checkThrow(gamma, delta, ft, false, pp, Type.negativeArraySizeException, S, P);
76         ft.checkSame(ft.stackPop(), Type.integer);
77         ft.stackPush(Type.arrayOf(cp.getClassIdent(c.getArgument(pp))));
78         break;
79
80     case 190: // arraylength
81         checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
82         ft.checkSub(ft.stackPop(), Type.botArray);
83         ft.stackPush(Type.integer);
84         break;
85
86     case 46: // iaload
87         checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
88         checkThrow(gamma, delta, ft, false, pp, Type.arrayIndexOutOfBoundsException, S, P);
89         ft.checkSame(ft.stackPop(), Type.integer);
90         ft.stackPush(Type.arrayBase(ft.checkSame(ft.stackPop(), Type.integerArray)));
91         break;
92
93     case 50: // aaload
94         checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
95         checkThrow(gamma, delta, ft, false, pp, Type.arrayIndexOutOfBoundsException, S, P);
96         ft.checkSame(ft.stackPop(), Type.integer);
97         ft.stackPush(Type.arrayBase(ft.checkSub(ft.stackPop(), Type.objectArray)));
98         break;
99
100    case 79: // iastore
101        checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
102        checkThrow(gamma, delta, ft, false, pp, Type.arrayIndexOutOfBoundsException, S, P);
103        ft.checkSame(ft.stackPop(), Type.integer);
104        ft.checkSame(ft.stackPop(), Type.integer);
105        ft.checkSame(ft.stackPop(), Type.integerArray);
106        break;
107

```

```

108     case 83: // aastore
109         checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
110         checkThrow(gamma, delta, ft, false, pp, Type.arrayIndexOutOfBoundsException, S, P);
111         {
112             String valueType = ft.checkSub(ft.stackPop(), Type.object);
113             ft.checkSame(ft.stackPop(), Type.integer);
114             ft.checkSuper(Type.arrayBase(ft.stackPop()), valueType);
115         }
116         break;
117
118     // CONSTANT POOL INSTRUCTIONS.
119
120     case 18: // ldc (ldc_w)
121         cp.getInteger(c.getArgument(pp)); // item must be integer within the constant pool
122         ft.stackPush(Type.integer);
123         break;
124
125     case 180: // getfield
126         checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
127         {
128             FieldRef fref = cp.getFieldRef(c.getArgument(pp));
129             ft.checkSub(ft.stackPop(), fref.getClassIdent());
130             ft.stackPush(fref.getType());
131         }
132         break;
133
134     case 181: // putfield
135         checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
136         {
137             FieldRef fref = cp.getFieldRef(c.getArgument(pp));
138             ft.checkSub(ft.stackPop(), fref.getType());
139             ft.checkSub(ft.stackPop(), fref.getClassIdent());
140         }
141         break;
142
143     case 182: // invokevirtual
144         {
145             MethRef mref = cp.getMethRef(c.getArgument(pp));
146             // Check arguments backwards
147             for (int i = mref.getArgumentCount()-1; i >= 0; --i)
148                 ft.checkSub(ft.stackPop(), mref.getArgumentType(i));
149             ft.checkSub(ft.stackPop(), mref.getClassIdent());
150             String rt = mref.getReturnType();
151             if (!Type.isVoid(rt)) ft.stackPush(rt);
152
153             // Check all potential throws that might be caught by a handler...
154             for (int i = 0; i < et.getExcCount(); ++i)

```

```

155         if (et.inTryRange(i, pp))
156             checkThrow(gamma, delta, ft, false, pp, et.getCatchType(i), S, P);
157     }
158     break;
159
160     case 187: // new
161         ft.stackPush(cp.getClassIdent(c.getArgument(pp)));
162         break;
163
164     case 192: // checkcast
165         checkThrow(gamma, delta, ft, false, pp, Type.classCastException, S, P);
166         {
167             String fromType = ft.stackPop();
168             String toType = cp.getType(c.getArgument(pp));
169             ft.stackPush(Type.compatible(ch, toType, fromType) ? fromType : toType);
170         }
171         break;
172
173     // JUMP INSTRUCTIONS.
174
175     case 153: // ifeq
176     case 154: // ifne
177     case 155: // iflt
178     case 156: // ifge
179     case 157: // ifgt
180     case 158: // ifle
181         ft.checkSame(ft.stackPop(), Type.integer);
182         checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
183         break;
184
185     case 159: // if_icmpeq
186     case 160: // if_icmpne
187     case 161: // if_icmplt
188     case 162: // if_icmpge
189     case 163: // if_icmpgt
190     case 164: // if_icmple
191         ft.checkSame(ft.stackPop(), Type.integer);
192         ft.checkSame(ft.stackPop(), Type.integer);
193         checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
194         break;
195
196     case 165: // if_acmpeq
197     case 166: // if_acmpne
198         ft.checkSub(ft.stackPop(), Type.object);
199         ft.checkSub(ft.stackPop(), Type.object);
200         checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
201         break;

```

```

202
203     case 198: // ifnull
204         ft.checkSub(ft.stackPop(), Type.object);
205         checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
206         break;
207
208     case 167: // goto
209         checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
210         ft.setTop();
211         break;
212
213     // ABRUPT INSTRUCTIONS.
214
215     case 172: // ireturn
216         ft.checkSame(ft.checkSame(ft.stackPop(), Type.integer), delta.getReturnType());
217         ft.setTop();
218         break;
219
220     case 176: // areturn
221         ft.checkSub(ft.checkSub(ft.stackPop(), Type.object), delta.getReturnType());
222         ft.setTop();
223         break;
224
225     case 177: // return
226         ft.setTop();
227         break;
228
229     case 191: // athrow
230         {
231             String cid_e = ft.stackPop();
232             //System.out.println(" athrow of " +cid_e);
233             boolean pr = (ch.compatible(Type.exception, cid_e)
234                 && !ch.compatible(Type.runtime, cid_e));
235             checkThrow(gamma, delta, ft, pr, pp, cid_e, S, P);
236             ft.setTop();
237         }
238         break;
239
240     default:
241         throw VerifError.getInstance("unknown_instruction_" + c.getOpcode(pp) + "_at_" + pp);
242     }
243 }
244
245 // Check single jump instruction in P and/or S.
246 static void checkJump(StdContext gamma, MethContext delta, int pp, int pp2,
247     FrameType ft, Saved S, Pending P)
248     throws VerifError

```

```

249     {
250         if (pp2 > pp)
251             P.set(pp2, ft);
252         else
253             ft.meetWith(S.get(pp2));
254     }
255
256     // Check handling of exception e thrown at program point pp.
257     static void checkThrow(StdContext gamma, MethContext delta,
258                          FrameType ft, boolean pr, int pp, String cid_e,
259                          Saved S, Pending P) throws VerifError
260     {
261         // Check local handlers first.
262         ClassHier ch = gamma.getClassHier();
263         ExcTable et = delta.getExcTable();
264         for (int i = 0; i < et.getExcCount(); ++i) {
265             if (et.inTryRange(i, pp)) {
266                 String cid = et.getCatchType(i);
267                 boolean definite = (ch.compatible(cid, cid_e));
268                 boolean potential = (ch.compatible(cid_e, cid));
269                 if (definite || potential) {
270                     FrameType fte = new FrameType(ft);
271                     fte.stackSet(cid_e);
272                     checkJump(gamma, delta, pp, et.getCatchPP(i), fte, S, P);
273                     if (definite) return;
274                 }
275             }
276         }
277
278         // Check method exception table for checked exceptions.
279         if (pr) {
280             ExcAtt ea = delta.getExcAtt();
281             ea.checkContains(cid_e);
282         }
283     }
284
285 }

```

Remark 7.3.13 (extensions). We have extended the program with a few instructions:

- The conditional instructions ifeq, iflt, ifge, and ifgt, are just like ifne.
- The two-parameter conditional instructions (if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, and if_acmpne) just require two values on the stack.

7.4 The Infrastructure Code

This section contains the main code which enables the use of the rule implementations to actually lightweight verify a standard Java class file.

Source 7.4.1 (LBV.java). Implements main program for lightweight verification of a lightweight certified class file.

```

1  class LBV {
2
3     // Lightweight bytecode verify class file named by argument.
4     // Usage: "LBV [-v] class [method]"
5     // "-v" option prints verbose messages to System.out.
6     // "class" argument names class (without .class suffix).
7     // "method" argument only method to verify.
8     // Class hierarchy is read from "class.ch" and certificates from "class.method.cert".
9     public static void main(String[] argv) {
10        boolean failed = false;
11
12        // Extract arguments or fail.
13        boolean verbose = (argv.length >= 1 && argv[0].equals("-v"));
14        int offset = (verbose ? 1 : 0);
15        String className = (argv.length > offset ? argv[offset] : null);
16        String onlyMethod = (argv.length > offset+1 ? argv[offset+1] : null);
17        if (className == null || argv.length > offset+2) {
18            System.err.println("Usage: java LBV [-v] class [method]");
19            System.exit(2);
20        }
21
22        if (verbose) System.out.println("LIGHTWEIGHT_BYTECODE_VERIFICATION_of_" + className + " ");
23
24        // Verification failure context.
25        try {
26            ClassInfo ci = BCEL.getClassInfo(className); // only system dependency!
27
28            // Top-level context.
29            StdContext gamma = ci.getStdContext();
30            if (verbose) gamma.getClassHier().printClassHier("\nCLASS_HIERARCHY.ch=");
31
32            // Verify methods one by one.
33            for (int i = 0; i < ci.getMethodCount(); ++i) {
34                ci.setMethodIndex(i);
35                String method = ci.getMethodName();
36                try {
37                    if (onlyMethod == null || onlyMethod.equals(method)) {
38
39                        if (verbose) System.out.println("\nMETHOD_" + ci.getMethodName() + " ");
40

```

```

41         Cert cert = ci.getCert();
42         if (verbose) cert.printCert("CERTIFICATE_␣ce_␣=");
43
44         if (verbose) System.out.println("VERIFICATION_␣fta_␣=");
45         Method m = ci.getMethod();
46         MethLbv.verify(gamma, m, cert, verbose);
47         if (verbose) System.out.println("VERIFICATION_OF_␣" + method + " _OK.");
48     }
49 }
50 catch (VerifError e) {
51     if (verbose)
52         System.out.println("VERIFICATION_OF_␣" + method + " _FAILED:");
53     System.err.println(method + " :_␣" + e.getMessage());
54 }
55 }
56 }
57 catch (VerifError e) {
58     System.out.println("Initialization_␣failed_␣(" + e.getMessage() + ").");
59     failed = true;
60 }
61
62     if (failed) System.exit(1);
63 }
64
65 }

```

Source 7.4.2 (ClassInfo.java). Datastructure for main program.

```

1  interface ClassInfo {
2
3      // Verification context.
4      StdContext getStdContext();
5
6      // Number of methods in the class.
7      int getMethodCount();
8
9      // Set "current" method index.
10     // Throws VerifError if the index is out of bounds 0..getMethodCount()-1.
11     void setMethodIndex(int index) throws VerifError;
12
13     // Get name of method with the set index.
14     String getMethodName();
15
16     // Get internal information of method with the set index.
17     // Note: uses static storage so only use one at the time.
18     Method getMethod();
19
20     // Get certificate for method with the set index.

```

```

21 // Note: uses static storage so only use one at the time.
22 // Throws VerifError if there is no certificate for the method.
23 Cert getCert() throws VerifError;
24 }

```

Source 7.4.3 (BCEL.java). Class file access interface using Markus Dahm's ByteCode Engineering Library (BCEL) package [12] such that we can access standard Java .class files.

```

1 // Implementation of class and certificate access interfaces for method
2 // lightweight bytecode verification using Markus Dahm's BCEL (Byte Code
3 // Engineering Library).
4
5 import java.io.*;
6 import java.util.Vector;
7 import java.util.BitSet;
8
9 class BCEL implements ClassInfo, StdContext, ConstPool, ClassHier,
10 Method, MethSig, CodeAtt, Code, ExcTable, ExcAtt, Cert,
11 de.fub.bytecode.Constants // include bytecode Constants
12 {
13 // Singleton instance.
14
15 private static BCEL self;
16
17 // Creation factory method.
18 public static ClassInfo getClassInfo(String name) throws VerifError {
19     if (self == null) self = new BCEL();
20     self.instantiate(name);
21     return self;
22 }
23
24 // State.
25
26 // ClassInfo state
27 private int mCount; // number of methods
28 private int mIndex; // "current" method index
29
30 // StdContext state.
31 private String file; // class file name
32 private String cid; // class type
33 private de.fub.bytecode.classfile.JavaClass jc; // BCEL class file object.
34 private boolean certDirty; // whether certificate needs to be reread
35
36 // ConstPool state.
37 private de.fub.bytecode.classfile.ConstantPool jcp; // BCEL constant pool
38 private int jcpLength; // BCEL constant pool length
39 private FieldOrMethRef item; // cached BCEL constant pool item
40

```

```

41 // ClassHier state.
42 private String[] c; // class names
43 private int[] superIndex; // c[superIndex[i]] is the superclass of c[i], or bot.
44
45 // Method, MethSig, and CodeAtt state.
46 private de.fub.bytecode.classfile.Method jm; // cached BCEL method
47 private String desc; // cached method type description
48 private de.fub.bytecode.classfile.Code jcode; // cached BCEL code
49 private String[] jxs; // escaping exceptions
50 private de.fub.bytecode.classfile.CodeException[] jhs; // exception handlers
51
52 // Cert state.
53 private int[] certLabel;
54 private FrameTypeMap certFrameType;
55 private int certPendingSize;
56
57 // Private dummy constructor.
58 private BCEL() {}
59
60 // Private instantiator.
61 private void instantiate(String name) throws VerifError {
62
63     // Class name and file name.
64     if (name.startsWith("L") && name.endsWith(";")) {
65         file = name.substring(1, name.length()-1);
66     }
67     else {
68         file = name.replace('.', '/');
69     }
70
71     // Class Hierarchy (from file).
72     {
73         Vector cs = new Vector();
74         Vector supercs = new Vector();
75
76         cs.add(Type.object);
77         supercs.add(null);
78
79         // Read file.
80         try {
81             set_input(file + ".ch");
82             skip("{}");
83             char ch = next_nonspace();
84             if (ch != '}') {
85                 unread1(ch);
86                 do {
87                     cs.add(next_type());

```

```

88         skip("<:");
89         supercs.add(next_type());
90     } while ((ch = next_nonspace()) == ',');
91     }
92     if (ch != '}') throw new IOException("','_or_'_expected");
93 }
94 catch (IOException x) {
95     throw VerifError.getInstance("class_hierarchy_input_error_" + x.getMessage() + "");
96 }
97
98     // Construct c array of class names.
99     int clength = cs.size();
100    c = new String[clength];
101    for (int i = 0; i < clength; ++i) c[i] = (String) cs.elementAt(i);
102
103    // Construct superIndex array of indices.
104    int[] si = new int[clength];
105    for (int i = 0; i < clength; ++i) {
106        String supercsi = (String) supercs.elementAt(i);
107        si[i] = findIndex(supercsi);
108    }
109    superIndex = si;
110 }
111
112 // Read BCEL class object.
113 try {
114     //System.out.println(" Reading '" + file + ".class' file.");
115     jc = new de.fub.bytecode.classfile.ClassParser(file+".class").parse();
116 }
117 catch (IOException x) {
118     throw VerifError.getInstance("could_not_read_class_file_" + file+".class'_" + x.getMessage() + "");
119 }
120
121 // Get class name.
122 cid = Type.type("L" + jc.getClassName() + "");
123
124 // Initialize constant pool.
125 jcp = jc.getConstantPool();
126 jcpLength = jcp.getLength();
127 item = null;
128
129 // Initialize method information.
130 mCount = jc.getMethods().length;
131 mIndex = -1;
132 jm = null;
133 desc = null;
134 jcode = null;

```

```

135     jxs = null;
136     jhs = null;
137     certDirty = true;
138 }
139
140 // ClassInfo implementation.
141
142 public StdContext getStdContext() {
143     return this;
144 }
145
146 public int getMethodCount() {
147     return mCount;
148 }
149
150 public void setMethodIndex(int index) throws VerifError {
151     if (index < 0 || index > getMethodCount())
152         throw VerifError.getInstance("no method number " + index);
153     mIndex = index;
154     jm = jc.getMethods()[index];
155     desc = jm.getSignature();
156     jcode = jm.getCode();
157     de.fub.bytecode.classfile.ExceptionTable jet = jm.getExceptionTable();
158     if (jet == null)
159         jxs = new String[0];
160     else {
161         jxs = new String[jet.getNumberOfExceptions()];
162         int[] jxis = jet.getExceptionIndexTable();
163         for (int i = 0; i < jxs.length; ++i)
164             jxs[i] = Type.type(getClassIdent(jxis[i]));
165     }
166     jhs = jcode.getExceptionTable();
167     certDirty = true;
168 }
169
170 public String getMethodName() {
171     return jm.getName();
172 }
173
174 public Method getMethod() {
175     return this;
176 }
177
178 public Cert getCert() throws VerifError {
179     if (certDirty) {
180
181         // Open certificate file for method.

```

```

182     try {
183         set_input(file + "." + getMethodName() + ".cert");
184     }
185     catch (IOException x) {
186         // Fall-back: empty certificate.
187         certFrameType = new FrameTypeMap(this, 0);
188         certLabel = new int[0];
189         int certPendingSize = 3;
190         return this;
191     }
192
193     // Read certificate file for method.
194     try {
195         char ch;
196         skip("(");
197
198         // Read label list.
199         BitSet labels = new BitSet();
200         skip("{");
201         ch = next_nonspace();
202         if (ch != ',') {
203             unread1(ch);
204             do {
205                 //System.out.println(" Got label");
206                 labels.set(next_integer());
207             } while ((ch = next_nonspace()) == ',');
208         }
209         if (ch != ',') throw new IOException("'" + ch + "' expected");
210         int l = 0;
211         for (int i = 0; i < labels.length(); ++i) {
212             if (labels.get(i)) {
213                 ++l;
214                 //System.out.println(" Got label #" + l + ": " + i);
215             }
216         }
217         certLabel = new int[l];
218         for (int i = 0, j = 0; i < labels.length(); ++i)
219             if (labels.get(i)) {
220                 //System.out.println(" Got label #" + i);
221                 certLabel[j++] = i;
222             }
223
224         skip(",");
225
226         // Read FrameType "deltas".
227         Vector pps = new Vector();
228         Vector fts = new Vector();

```

```

229     skip("{");
230     ch = next_nonspace();
231     if (ch != '}') {
232         unread1(ch);
233         do {
234             pps.add(new Integer(next_integer()));
235             skip(":");
236             FrameType ft = new FrameType(this, getMaxStack(), getMaxLocals());
237             skip("(");
238
239             skip("(");
240             ch = next_nonspace();
241             if (ch != '}') {
242                 unread1(ch);
243                 do {
244                     ft.stackPush(next_type());
245                 } while ((ch = next_nonspace()) == ',');
246             }
247             if (ch != '}') throw new IOException("','_or_'_expected");
248             skip(",");
249
250             skip("(");
251             int i = 0;
252             ch = next_nonspace();
253             if (ch != '}') {
254                 unread1(ch);
255                 do {
256                     ft.localsSet(i++, next_type());
257                 } while ((ch = next_nonspace()) == ',');
258             }
259             if (ch != '}') throw new IOException("','_or_'_expected");
260             if (i != getMaxLocals())
261                 throw new IOException("incorrect_local_variable_type_numbers");
262
263             fts.add(ft);
264             skip(")");
265         } while ((ch = next_nonspace()) == ',');
266     }
267     if (ch != '}') throw new IOException("','_or_'_expected");
268
269     certFrameType = new FrameTypeMap(this, pps.size());
270     for (int i = 0; i < pps.size(); ++i)
271         certFrameType.set(((Integer) pps.get(i)).intValue(), (FrameType) fts.get(i));
272
273     skip(",");
274
275     certPendingSize = next_integer();

```

```

276
277     skip(""));
278 }
279 catch (IOException x) {
280     throw VerifError.getInstance("certificate_input_error_" + x.getMessage() + "");
281 }
282 certDirty = false;
283 }
284 return this;
285 }
286
287 // StdContext implementation.
288
289 public ConstPool getConstPool() {
290     return this;
291 }
292
293 public String getClassIdent() {
294     return cid;
295 }
296
297 public ClassHier getClassHier() {
298     return this;
299 }
300
301 // ConstPool implementation.
302
303 public int getConstPoolLength() {
304     return jcpLength;
305 }
306
307 public String getClassIdent(int index) throws VerifError {
308     if (index <= 0 || index >= jcpLength)
309         throw VerifError.getInstance("no_constant_pool_item#" + index);
310     try {
311         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
312         if (it.getTag() == CONSTANT_Class) {
313             de.fub.bytecode.classfile.ConstantClass cc =
314                 (de.fub.bytecode.classfile.ConstantClass) it;
315             de.fub.bytecode.classfile.ConstantUtf8 cu8 =
316                 (de.fub.bytecode.classfile.ConstantUtf8) jcp.getConstant(cc.getNameIndex());
317             String t = cu8.getBytes();
318             return Type.type(t.charAt(0) == '[' ? t : "L" + t + ";");
319         }
320         else
321             throw VerifError.getInstance("constant_pool_item_number_" + index + " is not a ClassIdent");
322     }

```

```

323     catch (java.lang.ClassFormatError x) {
324         throw VerifError.getInstance("illegal_item");
325     }
326 }
327
328 public FieldRef getFieldRef(int index) throws VerifError {
329     try {
330         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
331         if (it.getTag() == CONSTANT_Fieldref) {
332             if (item == null) item = new FieldOrMethRef();
333             item.setIndex(index);
334             return (FieldRef) item;
335         }
336         else
337             throw VerifError.getInstance("constant_pool_item_number_" + index + "is_not_a_FieldRef");
338     }
339     catch (java.lang.ClassFormatError x) {
340         throw VerifError.getInstance("illegal_item");
341     }
342 }
343
344 public MethRef getMethRef(int index) throws VerifError {
345     try {
346         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
347         if (it != null && it.getTag() == CONSTANT_Methodref) {
348             if (item == null) item = new FieldOrMethRef();
349             item.setIndex(index);
350             return (MethRef) item;
351         }
352         else
353             throw VerifError.getInstance("constant_pool_item_number_" + index + "is_not_a_MethRef");
354     }
355     catch (java.lang.ClassFormatError x) {
356         throw VerifError.getInstance("illegal_item");
357     }
358 }
359
360 public int getInteger(int index) throws VerifError {
361     try {
362         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
363         if (it.getTag() == CONSTANT_Integer)
364             return ((de.fub.bytecode.classfile.ConstantInteger) it).getBytes();
365         else
366             throw VerifError.getInstance("constant_pool_item_number_" + index + "is_not_an_Integer");
367     }
368     catch (java.lang.ClassFormatError x) {
369         throw VerifError.getInstance("no_constant_pool_item_number_" + index);

```

```

370     }
371 }
372
373 // not used
374 public String getString(int index) throws VerifError {
375     try {
376         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
377         if (it.getTag() == CONSTANT_String) {
378             int x = ((de.fub.bytecode.classfile.ConstantString) it).getStringIndex();
379             de.fub.bytecode.classfile.ConstantUtf8 cu8 =
380                 (de.fub.bytecode.classfile.ConstantUtf8) jcp.getConstant(x);
381             return cu8.getBytes();
382         }
383         else
384             throw VerifError.getInstance("constant_pool_item_number_ " + index + " is not a String");
385     }
386     catch (java.lang.ClassFormatError x) {
387         throw VerifError.getInstance("no_constant_pool_item_number_ " + index);
388     }
389 }
390
391 public String getType(int index) throws VerifError {
392     if (index <= 0 || index >= jcpLength)
393         throw VerifError.getInstance("no_constant_pool_item_#" + index);
394     try {
395         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
396         if (it.getTag() == CONSTANT_Utf8) {
397             return Type.type("L" + ((de.fub.bytecode.classfile.ConstantUtf8) it).getBytes() + ";");
398         }
399         else if (it.getTag() == CONSTANT_Class) {
400             return Type.type(getClassIdent(index));
401         }
402         else
403             throw VerifError.getInstance("constant_pool_item_number_ " + index + " is not a Type");
404     }
405     catch (java.lang.ClassFormatError x) {
406         throw VerifError.getInstance("illegal_item_(" + x.getMessage() + ")");
407     }
408 }
409
410 // String representation of ConstPool item.
411 public String item(int index) {
412     StringBuffer it = new StringBuffer();
413     try {
414         de.fub.bytecode.classfile.Constant item = jcp.getConstant(index);
415         if (item == null)
416             return null;

```

```

417     else if (item.getTag() == CONSTANT_Class)
418         it.append(getClassIdent(index));
419     else if (item.getTag() == CONSTANT_Integer)
420         it.append(getInteger(index));
421     else if (item.getTag() == CONSTANT_String)
422         it.append("\'" + getString(index) + "\'");
423     else if (item.getTag() == CONSTANT_Fieldref) {
424         FieldRef fref = getFieldRef(index);
425         it.append("fieldref("+fref.getClassIdent()+", "+fref.getIdent()+", "+fref.getType()+")");
426     }
427     else if (item.getTag() == CONSTANT_Methodref) {
428         MethRef mref = getMethRef(index);
429         it.append("methodref("+mref.getClassIdent()+", "+methsig("+mref.getIdent()+", ");
430         String lead = "";
431         for (int a = 0; a < mref.getArgumentCount(); ++a) {
432             it.append(lead+mref.getArgumentType(a));
433             lead = ", ";
434         }
435         it.append(")], "+mref.getReturnType()+")");
436     }
437     else
438         return null;
439 }
440 catch (VerifError x) {
441     it.append("?" + index);
442 }
443 return it.toString();
444 }
445
446 // FieldRef and MethRef implementation.
447 class FieldOrMethRef implements FieldRef, MethRef {
448
449     // State.
450     private de.fub.bytecode.classfile.ConstantCP it;
451     private de.fub.bytecode.classfile.ConstantNameAndType nt;
452     private String sig;
453     void setIndex(int index) {
454         it = (de.fub.bytecode.classfile.ConstantCP) jcp.getConstant(index);
455         nt = (de.fub.bytecode.classfile.ConstantNameAndType)
456             jcp.getConstant(it.getNameAndTypeIndex());
457         sig = Type.type(((de.fub.bytecode.classfile.ConstantUtf8)
458             jcp.getConstant(nt.getSignatureIndex())).getBytes());
459     }
460 }
461
462 public String getClassIdent() {
463     int x = ((de.fub.bytecode.classfile.ConstantClass)

```

```

464         jcp.getConstant(it.getClassIndex()).getNameIndex();
465     return Type.type("L" + ((de.fub.bytecode.classfile.ConstantUtf8)
466         jcp.getConstant(x)).getBytes() + ";"");
467     }
468
469     public String getIdent() {
470         return Type.type(((de.fub.bytecode.classfile.ConstantUtf8)
471         jcp.getConstant(nt.getNameIndex()).getBytes());
472     }
473
474     public String getType() {
475         return sig;
476     }
477
478     public String getReturnType() {
479         return Type.type(de.fub.bytecode.generic.Type.getReturnType(sig).getSignature());
480     }
481
482     public int getArgumentCount() {
483         return de.fub.bytecode.generic.Type.getArgumentTypes(sig).length;
484     }
485
486     public String getArgumentType(int i) {
487         return Type.type(de.fub.bytecode.generic.Type.getArgumentTypes(sig)[i].getSignature());
488     }
489 }
490
491 // ClassHier implementation.
492
493 public boolean isClass(String type) {
494     return (findIndex(type) >= 0);
495 }
496 int findIndex(String type) {
497     for (int i = 0; i < c.length; ++i) {
498         if (c[i] == type) {
499             return i;
500         }
501     }
502     return -1;
503 }
504
505 public boolean compatible(String cid, String sub_cid) throws VerifError {
506     int i = findIndex(cid);
507     if (i < 0)
508         throw VerifError.getInstance("no_class_␣" + cid + "␣in_class_hierarchy");
509     int sub_i = findIndex(sub_cid);
510     if (sub_i < 0)

```

```

511         throw VerifError.getInstance("no_class_" + sub_cid + "_in_class_hierarchy");
512         while (sub_i != i && sub_i != -1)
513             sub_i = superIndex[sub_i];
514         return (sub_i == i);
515     }
516
517     public String meet(String cid1, String cid2) throws VerifError {
518         int i1 = findIndex(cid1);
519         if (i1 < 0)
520             throw VerifError.getInstance("no_class_" + cid1 + "_in_class_hierarchy");
521         int i2 = findIndex(cid2);
522         if (i2 < 0)
523             throw VerifError.getInstance("no_class_" + cid2 + "_in_class_hierarchy");
524         while (i1 > 0) {
525             for (int i = i2; i > 0; i = superIndex[i])
526                 if (i1 == i) return c[i];
527             i1 = superIndex[i1];
528         }
529         return Type.object;
530     }
531
532     // String representation of ClassHier.
533     public void printClassHier(String prefix) {
534         for (int i = 0; i < c.length; ++i) {
535             int si = superIndex[i];
536             System.out.println((i == 0 ? prefix+"_{ " : "  ") + c[i]+ " <:" + (si < 0 ? "bot" : c[si]));
537         }
538         System.out.println("}");
539     }
540
541     // Method implementation.
542
543     public MethSig getMethSig() {
544         return this;
545     }
546
547     public String getReturnType() {
548         return Type.type(de.fub.bytecode.generic.Type.getReturnType(desc).getSignature());
549     }
550
551     public CodeAtt getCodeAtt() {
552         return this;
553     }
554
555     public ExcAtt getExcAtt() {
556         return this;
557     }

```

```
558
559 // MethSig implementation.
560
561 public String getIdent() {
562     return jm.getName();
563 }
564
565 public int getArgumentCount() {
566     return de.fub.bytecode.generic.Type.getArgumentTypes(desc).length;
567 }
568
569 public String getArgumentType(int i) {
570     return Type.type(de.fub.bytecode.generic.Type.getArgumentTypes(desc)[i].getSignature());
571 }
572
573 // CodeAtt implementation.
574
575 public int getMaxStack() {
576     return jcode.getMaxStack();
577 }
578
579 public int getMaxLocals() {
580     return jcode.getMaxLocals();
581 }
582
583 public Code getCode() {
584     return this;
585 }
586
587 public ExcTable getExcTable() {
588     return this;
589 }
590
591 // Code implementation.
592
593 public int getMaxCode() {
594     return jcode.getCode().length - 1;
595 }
596
597 public int getOpcode(int pp) {
598     int op = jcode.getCode()[pp];
599     if (op < 0) op += 256;
600     switch (op) {
601
602     case WIDE:
603         return getOpcode(pp+1);
604
```

```
605     case BIPUSH:
606     case SIPUSH:
607     case ICONST_M1:
608     case ICONST_2:
609     case ICONST_3:
610     case ICONST_4:
611     case ICONST_5:
612         return ICONST_0;
613
614     case ILOAD_0:
615     case ILOAD_1:
616     case ILOAD_2:
617     case ILOAD_3:
618         return ILOAD;
619
620     case ALOAD_0:
621     case ALOAD_1:
622     case ALOAD_2:
623     case ALOAD_3:
624         return ALOAD;
625
626     case ISTORE_0:
627     case ISTORE_1:
628     case ISTORE_2:
629     case ISTORE_3:
630         return ISTORE;
631
632     case ASTORE_0:
633     case ASTORE_1:
634     case ASTORE_2:
635     case ASTORE_3:
636         return ASTORE;
637
638     case LDC_W:
639         return LDC;
640
641     default:
642         return op;
643     }
644 }
645
646 public int getLength(int pp) {
647     int op = jcode.getCode()[pp];
648     if (op < 0) op += 256;
649     switch (op) {
650
651     case WIDE:
```

```

652         return 1 + getLength(pp+1);
653
654     case NEW:
655         // Special case: new = NEW + DUP + INVOKESPECIAL <init> assumed!
656         return 7;
657
658     case LDC_W:
659     case GETFIELD:
660     case PUTFIELD:
661     case INVOKEVIRTUAL:
662     case ANEWARRAY:
663     case CHECKCAST:
664     case IFEQ:
665     case IFNE:
666     case IFLT:
667     case IFGT:
668     case IFLE:
669     case IFGE:
670     case GOTO:
671     case IFNULL:
672     case IFNONNULL:
673     case SIPUSH:
674         return 3;
675
676     case ILOAD:
677     case ALOAD:
678     case ISTORE:
679     case ASTORE:
680     case BIPUSH:
681     case NEWARRAY:
682     case LDC:
683         return 2;
684
685     default:
686         return 1;
687     }
688 }
689
690 public int getArgument(int pp) {
691     int op = jcode.getCode()[pp];
692     if (op < 0) op += 256;
693
694     int b1, b2; // scratch variables
695
696     switch (op) {
697
698     case WIDE:

```

```
699     ++pp; // hack: use 2-byte unsigned argument code below...
700     case LDC_W:
701     case GETFIELD:
702     case PUTFIELD:
703     case INVOKEVIRTUAL:
704     case NEW:
705     case ANEWARRAY:
706     case CHECKCAST:
707         // Instructions with a two-byte unsigned argument.
708         b1 = jcode.getCode()[pp+1];
709         if (b1 < 0) b1 +=256;
710         b2 = jcode.getCode()[pp+2];
711         if (b2 < 0) b2 +=256;
712         return (b1*256 + b2);
713
714     case IFEQ:
715     case IFNE:
716     case IFLT:
717     case IFGT:
718     case IFLE:
719     case IFGE:
720     case GOTO:
721     case IFNULL:
722     case IFNONNULL:
723         // Instructions with a two-byte signed argument.
724         b1 = jcode.getCode()[pp+1];
725         b2 = jcode.getCode()[pp+2];
726         if (b2 < 0) b2 +=256;
727         return (b1*256 + b2);
728
729     case LDC:
730     case ILOAD:
731     case ALOAD:
732     case ISTORE:
733     case ASTORE:
734     case NEWARRAY:
735         // Instructions with a single-byte unsigned argument.
736         b1 = jcode.getCode()[pp+1];
737         if (b1 < 0) b1 +=256;
738         return b1;
739
740         // Instructions with implied argument.
741     case ILOAD_0:
742     case ALOAD_0:
743     case ISTORE_0:
744     case ASTORE_0:
745         return 0;
```

```

746
747     case ILOAD_1:
748     case ALOAD_1:
749     case ISTORE_1:
750     case ASTORE_1:
751         return 1;
752
753     case ILOAD_2:
754     case ALOAD_2:
755     case ISTORE_2:
756     case ASTORE_2:
757         return 2;
758
759     case ILOAD_3:
760     case ALOAD_3:
761     case ISTORE_3:
762     case ASTORE_3:
763         return 3;
764
765     // The default is instructions with no offset/branch argument.
766     default:
767         return 0;
768     }
769 }
770
771 // String representation of byte code instruction.
772 public String instruction(int pp) {
773     StringBuffer ins = new StringBuffer();
774     int opcode = getOpcode(pp);
775     int arg = getArgument(pp);
776
777     switch (opcode) {
778
779     case WIDE:
780         switch (getOpcode(pp+1)) {
781             case ILOAD: ins.append("wide_1load_1"+arg); break;
782             case ALOAD: ins.append("wide_1aload_1"+arg); break;
783             case ISTORE: ins.append("wide_1istore_1"+arg); break;
784             case ASTORE: ins.append("wide_1astore_1"+arg); break;
785         }
786         break;
787
788     case ACONST_NULL: ins.append("aconst_null"); break;
789     case ICONST_M1: ins.append("iconst_-1"); break;
790     case ICONST_0: ins.append("iconst_0"); break;
791     case ICONST_1: ins.append("iconst_1"); break;
792     case ICONST_2: ins.append("iconst_2"); break;

```

```

793     case ICONST_3: ins.append("iconst_3"); break;
794     case ICONST_4: ins.append("iconst_4"); break;
795     case ICONST_5: ins.append("iconst_5"); break;
796     case POP: ins.append("pop"); break;
797     case DUP: ins.append("dup"); break;
798     case IADD: ins.append("iadd"); break;
799     case ISUB: ins.append("isub"); break;
800     case IRETURN: ins.append("ireturn"); break;
801     case ARETURN: ins.append("areturn"); break;
802     case RETURN: ins.append("return"); break;
803     case ATHROW: ins.append("athrow"); break;
804     case ARRAYLENGTH: ins.append("arraylength"); break;
805     case AASTORE: ins.append("aastore"); break;
806     case AALOAD: ins.append("aaload"); break;
807
808     case ILOAD: ins.append("iload_" + arg); break;
809     case ALOAD: ins.append("aload_" + arg); break;
810     case ISTORE: ins.append("istore_" + arg); break;
811     case ASTORE: ins.append("astore_" + arg); break;
812     case BIPUSH: ins.append("bipush_" + arg); break;
813     case SIPUSH: ins.append("sipush_" + arg); break;
814
815     case LDC: ins.append("ldc_" + item(arg)); break;
816     case LDC_W: ins.append("ldc_w_" + item(arg)); break;
817     case GETFIELD: ins.append("getfield_" + item(arg)); break;
818     case PUTFIELD: ins.append("putfield_" + item(arg)); break;
819     case INVOKEVIRTUAL: ins.append("invokevirtual_" + item(arg)); break;
820     case NEW: ins.append("new_" + item(arg)); break;
821     case NEWARRAY: ins.append("newarray_" + item(arg)); break;
822     case ANEWARRAY: ins.append("anewarray_" + item(arg)); break;
823     case CHECKCAST: ins.append("checkcast_" + item(arg)); break;
824
825     case IFEQ: ins.append("ifeq_" + (arg < 0 ? "" : "+") + arg); break;
826     case IFNE: ins.append("ifne_" + (arg < 0 ? "" : "+") + arg); break;
827     case IFLT: ins.append("iflt_" + (arg < 0 ? "" : "+") + arg); break;
828     case IFGT: ins.append("ifgt_" + (arg < 0 ? "" : "+") + arg); break;
829     case IFLE: ins.append("ifle_" + (arg < 0 ? "" : "+") + arg); break;
830     case IFGE: ins.append("ifge_" + (arg < 0 ? "" : "+") + arg); break;
831     case GOTO: ins.append("goto_" + (arg < 0 ? "" : "+") + arg); break;
832     case IFNULL: ins.append("ifnull_" + (arg < 0 ? "" : "+") + arg); break;
833     case IFNONNULL: ins.append("ifnonnull_" + (arg < 0 ? "" : "+") + arg); break;
834
835     default: ins.append("UNKNOWN_" + opcode + "_" + arg); break;
836 }
837 return ins.toString();
838 }
839

```

```

840 // ExcAtt implementation.
841
842 public void checkContains(String except) throws VerifError {
843     for (int i = 0; i < jxs.length; ++i)
844         if (compatible(jxs[i], except))
845             return;
846     throw VerifError.getInstance("exception_unhandled");
847 }
848
849 // ExcTable implementation.
850
851 public int getExcCount() {
852     return jhs.length;
853 }
854
855 public boolean inTryRange(int n, int pp) {
856     return (jhs[n].getStartPC() <= pp && pp < jhs[n].getEndPC());
857 }
858
859 public String getCatchType(int n) throws VerifError {
860     String t = Type.type(getClassIdent(jhs[n].getCatchType()));
861     //System.out.println(" getCatchType(" +n+ ") = " +t);
862     return t;
863 }
864
865 public int getCatchPP(int n) {
866     return jhs[n].getHandlerPC();
867 }
868
869 // Cert implementation.
870
871 public boolean isLabel(int pp) {
872     for (int i = 0; i < certLabel.length; ++i) {
873         if (certLabel[i] == pp) return true;
874     }
875     return false;
876 }
877
878 public int labelCount() {
879     return certLabel.length;
880 }
881
882 public void applyDeltaTo(FrameType ft, int pp) throws VerifError {
883     if (certFrameType.contains(pp))
884         ft.meetWith(certFrameType.get(pp));
885 }
886

```

```

887 public int getPendingSize() {
888     return certPendingSize;
889 }
890
891 public void printCert(String prefix) {
892     StringBuffer sb = new StringBuffer(prefix+"({}");
893     for (int i = 0; i < certLabel.length; ++i)
894         sb.append((i==0?"":","") + certLabel[i]);
895     sb.append("},");
896     System.out.println(sb.toString());
897     System.out.println("└─"+certFrameType.frameTypeMap()+"");
898     System.out.println("└─"+certPendingSize+"");
899 }
900
901 // Input utility methods.
902
903 // State.
904 private InputStreamReader input; // input stream
905 private int unread_char; // lookahead char
906
907 // Set input source.
908 private void set_input(String name) throws IOException {
909     //System.out.println(" Input file '" + name + "':");
910     input = new FileReader(name);
911     unread_char = -1;
912 }
913
914 // Return next character.
915 private char next_char() throws IOException {
916     char ch = read1();
917     //System.out.println(" read '" + ch + "'");
918     return ch;
919 }
920
921 // Return next non-space character.
922 private char next_nonspace() throws IOException {
923     char ch = read1();
924     while (Character.isWhitespace(ch)) ch = read1();
925     //System.out.println(" read '" + ch + "'");
926     return ch;
927 }
928
929 // Skip spaces and s.
930 private void skip(String s) throws IOException {
931     unread1(read1nonspace()); // hack to skip spaces before reading
932     for (int i = 0; i < s.length(); ++i) {
933         char ch = read1();

```

```

934     if (ch != s.charAt(i)) throw new IOException("'" + s + "' expected");
935     }
936     //System.out.println(" skip '" + s + "'");
937 }
938
939 // Return next integer on the input.
940 private int next_integer() throws IOException {
941     int i = 0;
942     char ch;
943     for (ch = read1nonspace(); Character.isDigit(ch); ch = read1()) {
944         i = i*10 + Character.digit(ch, 10);
945     }
946     unread1(ch);
947     //System.out.println(" integer '" + i + "'");
948     return i;
949 }
950
951 // Return next (extended) type identifier.
952 private String next_type() throws IOException {
953     StringBuffer sb = new StringBuffer();
954     char ch = read1nonspace();
955     if (ch == '[') {
956         sb.append(ch);
957         ch = read1();
958     }
959     if (ch == 'I') {
960         sb.append('I');
961     }
962     else if (ch == 'b') {
963         unread1(ch);
964         skip("bot");
965         sb.append("bot");
966     }
967     else if (ch == 't') {
968         unread1(ch);
969         skip("top");
970         sb.append("bot");
971     }
972     else if (ch == 'L') {
973         sb.append(ch);
974         ch = read1();
975         while (Character.isJavaIdentifierStart(ch)) {
976             do {
977                 sb.append(ch);
978                 while (Character.isJavaIdentifierPart(ch = read1())) sb.append(ch);
979             } while (ch == '/');
980             if (ch != ';') throw new IOException("';' expected");

```

```

981         sb.append(ch);
982     }
983 }
984     else throw new IOException("JVM_type_or_'bot'_or_'top'_expected");
985
986     return Type.type(sb.toString());
987 }
988
989 // Low-level character read.
990 private char read1() throws IOException {
991     if (unread_char >= 0) {
992         char ch = (char) unread_char;
993         unread_char = -1;
994         return ch;
995     }
996     else {
997         int ch = input.read();
998         if (ch < 0)
999             throw new IOException("end_of_file");
1000         else {
1001             return (char) ch;
1002         }
1003     }
1004 }
1005
1006 // Low-level non-space character read.
1007 private char read1nonspace() throws IOException {
1008     char ch = read1();
1009     while (Character.isWhitespace(ch)) ch = read1();
1010     return ch;
1011 }
1012
1013 // Low-level character unread.
1014 private void unread1(char ch) {
1015     unread_char = ch;
1016 }
1017
1018 }

```

Remark 7.4.4 (BCEL hacks). The `BCEL.java` file includes several hacks to make the use of standard classfiles feasible in connection with our subset:

- A new instruction in the class file is assumed followed by `dup` and an `invokespecial` of the constructor `<init>` method; the whole lot generates just a single new instruction.
- Several alternative forms of integer load instructions are “coerced” into `iconst_0` instructions.

7.5 The User Manual

Documentation 7.5.1 (Installation). In order to use the program you need the following:

- A Java 2 standard edition execution environment,
- `lbv.jar` file accompanying this thesis, and
- `bcel.jar` file with Markus Dahm’s Byte Code Engineering Library [12].

Once these are available you can run the lightweight bytecode verifier by placing `lbv.jar` and `bcel.jar` in the same directory, say *d*, and run

```
java -jar d/lbv.jar arguments
```

where the possible *arguments* are given in the usage below.

Documentation 7.5.2 (Usage). The program takes the following *arguments* (in that sequence):

-v Indicates that `lbv` should output the derived frame type as it goes. *Optional*.

class The class to verify (without `.class` extension).

method Method name to verify (by default all methods are verified). *Optional*.

In addition it expects to find these files:

class.ch File describing class hierarchy for class context. Includes a `{}`-delimited, `,`-separated list of simple direct superclass relations using the JVM notation for class names and written “*subclass <: superclass*”.

class.method.cert for each method: File with certificate for that *method* in the *class*. The certificate is expected as a triple of

- the labels as a `{}`-delimited, `,`-separated list of program point integers,
- the frame type certificate as a `{}`-delimited, `,`-separated list of simple maps of the form “*program-point : frame-type*” where a *frame-type* in turn has the form “*< stack-type , locals-type >*” with both of *stack-type* and *locals-type* a `.`-separated list of extended *type-name* identifiers from the set `I, bot, top, Lclass;`, and `[type-name,` and
- the largest size of `P` during the proof.

Example 7.5.3 (Gcd11.ch class hierarchy file). A possible class hierarchy file corresponding to the class hierarchy used by the `cksum()` example is the following.

```
{
Ljava/lang/Throwable; <: Ljava/lang/Object; ,
Ljava/lang/Exception; <: Ljava/lang/Throwable; ,
Ljava/lang/RuntimeException; <: Ljava/lang/Exception; ,
Ljava/lang/NullPointerException; <: Ljava/lang/RuntimeException; ,
```

```
Ljava/lang/ArrayIndexOutOfBoundsException; <: Ljava/lang/RuntimeException; ,
Ljava/lang/ArrayStoreException; <: Ljava/lang/RuntimeException; ,
Ljava/lang/NegativeArraySizeException; <: Ljava/lang/RuntimeException; ,
Ljava/lang/ClassCastException; <: Ljava/lang/RuntimeException; ,
LCrCardRd; <: Ljava/lang/Object; ,
LUnsetCrCard; <: Ljava/lang/Exception; ,
LAbort; <: Ljava/lang/Exception; ,
LChecksum; <: Ljava/lang/Object; ,
LGcd11; <: LChecksum;
}
```

Example 7.5.4 (Gcd11.cksum.cert certificate file). A possible certificate file for the `cksum()` example:

```
({20}, {}, 2)
```

Chapter 8

Comparisons

In this chapter, we compare lightweight verification technique as presented in Chapter 5 with recently implemented bytecode verification techniques for limited devices: In Section 8.1 we explain to what extent lightweight verification is a generalization of Sun’s J2ME “KVM” Java Virtual Machine, and in Section 8.2 we show how lightweight bytecode verification is a generalization of Leroy’s proposed “On-card Verifier”.

8.1 Sun’s “StackMap” Attribute

Sun’s J2ME [56] “Connected, Limited Device Configuration” specification [58] defines the requirements to a J2ME Java Virtual Machine in general and the classfile (bytecode) verifier in particular. The latter has been implemented by Sheng Liang [28] for the “KVM” virtual machine [59] using a “Stack map attribute” that is a slight simplification of our certificate FTC component: The KVM verifier simplifies the lightweight verification certificate by having no label set but containing in FTC the $FTA(PP)$ for *all* PP that are the target of a jump whether forward or backward. This makes it possible to check all constraints immediately, even for forward jumps, by comparing to the frame type stored in the certificate.

Example 8.1.1 (cksum() stack map attribute). The jump pattern of `cksum()` as illustrated in Figure 4.5 corresponds to a stack map corresponding to the certificate

$$(8.1.1a) \quad CE = \langle \{ PP \mapsto FTA(PP) \mid PP \in \{10, 15, 20, 39, 47\} \}, \emptyset \rangle$$

Below we modify the rules of LBV to include this simplified view.

Definition 8.1.2 (KVM simplifications). The rules of Chapter 5 should be modified as follows to simulate the KVM verifier:

1. For each *backward jump* rule (5.3.8a), (5.3.9a), (5.4.6a), and (5.4.7a), add a variant where $S(PP'')$ is replaced with $FTC(PP'')$ in the premise.
2. For each *forward jump* rule (5.3.8b), (5.3.9b), (5.4.6b), and (5.4.7b), add a variant with the premise $CH \vdash FTC(PP'') \sqsubseteq FT_{PP'}$ added and the side condition updating P removed (replacing occurrences of P' with P).

Note that this creates four rules per jump category.

Theorem 8.1.3 (Simulation of KVM). *The system of Definition 8.1.2 simulates the KVM verifier.*

Proof. Assume bytecode packaged with certificate $CE = \langle \text{FTC}, \epsilon \rangle$ where FTC maps all target program points. Clearly S will never be extended thus none of the original backward rules will be invoked. Similarly, we can always choose to use the new variant for forward rules thus we never need to extend P . Thus we can simulate the KVM's bytecode verifier's linear run through the code with only the single "current" frame type being updated throughout. \square

Remark 8.1.4 (combining KVM with lightweight bytecode verification). Clearly one could combine the above KVM rules with the lightweight verifier, however, this would break the uniqueness of certificates.

Remark 8.1.5 (resource use of KVM's bytecode verification). KVM's bytecode verifier stores a larger certificate, however, the certificate is immutable and can thus be stored in flash memory in contrast to our approach where S and P must be stored in scratch memory. In practice it is often worthwhile to trade the overhead of downloading the larger certificate for using less scratch memory – indeed the KVM algorithm derived from the above modified lightweight bytecode system only uses the scratch memory occupied by the current frame type.

8.2 Leroy's "On-Card Verifier"

Leroy proposes a collection of (automizable) simplifications of bytecode that makes it possible to use an "On-Card Verifier" [27] that requires very few resources. In this section we show how our certificate collapses to nothing with Leroy's constraints thus making lightweight bytecode verification mimic Leroy's algorithm in that case.

Definition 8.2.1 (Leroy's constraints). Leroy operates with the following constraints:

1. From a certain point PP_0 all following frame types have a constant local variable type. Formally, $\exists LT: \forall PP, PP \geq PP_0: \text{FTA}(PP) = \langle \text{ST}_{PP}, LT \rangle$.
2. Every jump target PP satisfies $PP \geq PP_0$ and $\text{FTA}(PP) = \langle \epsilon, LT \rangle$ (with PP_0 and LT as above).

Example 8.2.2 (cksum() transformed to conform to Leroy's constraints). Figure 8.1 shows our `cksum()` example after it has been transformed to conform to Leroy's restrictions. It has $PP_0 = 9$, $LT = \text{Gcd11} \cdot \text{CrCardRd} \cdot \text{int} \cdot \text{int} \cdot \text{int}$, and the stack type is ϵ for each target program point (in $\{19, 24, 29, 48, 56\}$).

The above considerations leads to the following:

Theorem 8.2.3 (Simulation of Leroy's algorithm). *Code satisfying the restrictions of Definition 8.2.1 can be equipped with a certificate of the form $\langle \epsilon, LS \rangle$ such that lightweight verification succeeds.*

PP	FTA _{pp} .ST	FTA _{pp} .LT (this·ccnum·x·y·z)	I
0	ε	Gcd11·CrCardRd·⊥·⊥·⊥	iconst_0
1	int	Gcd11·CrCardRd·⊥·⊥·⊥	istore[2]
3	ε	Gcd11·CrCardRd·int·⊥·⊥	iconst_0
4	int	Gcd11·CrCardRd·int·⊥·⊥	istore[3]
6	ε	Gcd11·CrCardRd·int·int·⊥	iconst_0
7	int	Gcd11·CrCardRd·int·int·⊥	istore[4]
9	ε	Gcd11·CrCardRd·int·int·int	aload[1]
11	CrCardRd	Gcd11·CrCardRd·int·int·int	invokevirtual[1]
14	int	Gcd11·CrCardRd·int·int·int	istore[2]
16	ε	Gcd11·CrCardRd·int·int·int	goto[+8]
19	UnsetCrCard	Gcd11·CrCardRd·int·int·int	pop
20	ε	Gcd11·CrCardRd·int·int·int	new[2]
23	Abort	Gcd11·CrCardRd·int·int·int	athrow
24	ε	Gcd11·CrCardRd·int·int·int	ldc_w[3]
27	int	Gcd11·CrCardRd·int·int·int	istore[3]
29	ε	Gcd11·CrCardRd·int·int·int	iload[2]
31	int	Gcd11·CrCardRd·int·int·int	iload[3]
33	int·int	Gcd11·CrCardRd·int·int·int	isub
34	int	Gcd11·CrCardRd·int·int·int	istore[4]
36	ε	Gcd11·CrCardRd·int·int·int	iload[4]
38	int	Gcd11·CrCardRd·int·int·int	ifle[+10]
41	ε	Gcd11·CrCardRd·int·int·int	iload[4]
43	int	Gcd11·CrCardRd·int·int·int	istore[2]
45	ε	Gcd11·CrCardRd·int·int·int	goto[-16]
48	ε	Gcd11·CrCardRd·int·int·int	iload[4]
50	int	Gcd11·CrCardRd·int·int·int	ifne[+6]
53	ε	Gcd11·CrCardRd·int·int·int	iload[2]
55	int	Gcd11·CrCardRd·int·int·int	ireturn
56	ε	Gcd11·CrCardRd·int·int·int	iload[2]
58	int	Gcd11·CrCardRd·int·int·int	istore[4]
60	ε	Gcd11·CrCardRd·int·int·int	iload[3]
62	int	Gcd11·CrCardRd·int·int·int	istore[2]
64	ε	Gcd11·CrCardRd·int·int·int	iload[4]
66	int	Gcd11·CrCardRd·int·int·int	istore[3]
68	ε	Gcd11·CrCardRd·int·int·int	goto[-39]

Figure 8.1: cksum() that verifies with Leroy's algorithm.

Proof. The constraints clearly imply that a certificate can have the form $\langle LS, \epsilon \rangle$, *i.e.*, has an empty FTC component, as jumps are always between identical frame types $\langle \epsilon, LT \rangle$. Thus the FTC component of any certificate will be empty. The certificate *may* contain an LS component but this corresponds to the fact that Leroy's algorithm permits testing for whether a program point is a "jump target". \square

Remark 8.2.4 (resource use with Leroy's restrictions). The lightweight proof system will construct S and P as usual but with the peculiarity that all the values are the same, namely $\langle \epsilon, LT \rangle$. So one can optimize an implementation to not actually store these values but merely the program points in each of S and T that have this value. This optimization corresponds to the space use of Leroy's algorithm where only the "current" frame type is stored along with information about which program points are jump targets.

Chapter 9

Java Access Protection through Typing

This chapter is a slightly extended reprint of a joint paper with Kristoffer Rose [49].

Abstract We propose integrating field access in general, and dedicated read-only field access in particular, into the Java type system. The principal gain is that “getter” methods can be eliminated such that

- fast static lookup can be used instead of dynamic dispatch for field access (without requiring a sophisticated inlining analyses),
- the (noticeable) space required by getter methods is avoided,
- denial-of-service attacks on field access is prevented, and
- access protection violations can be discovered by the bytecode verifier thus further simplifying the required run-time support.

We obtain this by extending a formalization of the Java bytecode verifier with access control so we can prove that the change is safe and backwards compatible.

9.1 Introduction

Object-oriented programming languages in general, and Java in particular, do not distinguish between read- and write-access to fields. Instead the recommended method to only permit read access to a field is to make the field private and write a “getter” method that accesses the field and returns the stored value.

For Java, the semantics of field access states that the actual field location accessed in an object can be determined statically (at compile-time), whereas the actual getter method invocation is determined dynamically (at run-time) [19, §15.10.1]. This has the following consequences:

- Using a getter method is significantly slower (at run-time) than using a direct field access. (The traditional remedy for this is to declare getter methods `final` which permits the compiler to *inline* its body, *i.e.*, insert the field access instruction directly at the invocation place. In Java this is frequently not feasible because Java employs *dynamic class loading* which

means that often a class to inline from is not available when installing a class using a getter method.)

- It is possible to access the field belonging to a particular (super)class of an object by simply casting the object of the field access to the appropriate class. One cannot obtain a similar effect with a getter method. (One may see this as a feature rather than an inconvenience.)
- “Denial-of-service” attacks are possible in that a getter method can be overridden by a subclass. (This can also be avoided by declaring the method `final`.)
- Finally, getter methods may add a significant space overhead to class files since they must be declared and their code given. For example, getter methods account for about one fourth of the total number of methods in the standard Java “`java.*`” package source classes.¹

Furthermore the *Java virtual machine* (JVM) specifies that field access control is performed through (dynamic) load and run time checks. This seems a shame since everything else about fields is static.

Here is a traditional example with a getter method: an object that simply contains an integer value that should be publicly readable.

```
class CrCardRd1 {
    int it;
    public int getIt() {return it;}
}
```

Access to the `it` field value of an object `cc` of type `CrCardRd1` requires the method invocation `cc.getIt()` with the problems discussed above.

In this paper we propose a simple modification in two steps that eliminates the problem altogether:

1. add a special *get-specific* access modifier that permits making the reading of a field “more public” than the modification of it, and
2. integrate field access checks into the type system.

In effect we propose replacing the above code with

```
class CrCardRd2 {
    read public int it;
}
```

which explicitly permits everyone to read off the field value with the usual field access syntax `cc.it` (but not to assign to it).

¹This measure obtained for Sun’s JDK 1.1 [54] with the unix commands “`find jdk1.1 -name '*.java' -exec grep ' +public .*(' '{}' ' ;' | wc -l`” to get the total number of public methods (4317), and “`find jdk1.1 -name '*.java' -exec egrep ' +public .* get.*(' '{}' ' ;' | wc -l`” to get the number of getter methods (999).

Accessibility from	Modifier	private	package	protected	public
		same class	✓	✓	✓
other class, same package	×	✓	✓	✓	
subclass outside package	×	×	✓	✓	
other class outside package	×	×	×	✓	

Table 9.1: Java Access Modifiers.

Plan. In section 9.2 we propose a minimal extension of the Java language [19] with the desired semantics, and since the Java runtime environment is centered around the JVM [30], we explain how the modification could be specified for the JVM. In section 9.3 we then explain how we can integrate field access into the type system of the JVM to be performed by the JVM *bytecode verifier*. Finally, we conclude in section 9.4 with some remarks on future work.

9.2 Read-only Field Access in Java

Recall that the Java access “modifiers” change the access rights as shown in table 9.1. (The “package” modifier is the default assumed when no modifier keyword is present and the table has ✓ when access is permitted and × when it is not.) Notice that the permissions are strictly included in each other and, in fact, statically checkable, since both the class hosting the field and the method attempting to access it are statically known.

We propose to extend the Java language with syntax for specifying an access modifier specific to *reading* a field value. This can be done with the following syntax extension to the Java Language Specification [19, §8.3.1]:

FieldDeclaration:

*FieldModifiers*_{op} *ReadModifier*_{op} *Type* *VariableDeclarators* ;

...

ReadModifier:

read *AccessModifier*_{op}

The semantics of the new construction is that we must separate field *access* from field *assignment*: an access that is not an assignment, *i.e.*, is not a Java *LeftHandSide* [19, §15.25], is permitted if either of the (original) field access modifier or the specific read access modifier (if any) permits it.

By using “either” we ensure that our extension is conservative in that old systems ignoring the read modifier will always be strictly less permissive than new ones.

In fact we can emulate the effect of the declaration

```
class c {
  w read r t f;
}
```

(with *c* a class name, *w* and *r* field modifiers, *t* a field type, and *f* a field name) by

```
class c {
  w t f;
  final r t get_c_f() {return f; }
}
```

where we must then replace all read accesses of the form *o.f* (for some object *o* with a static (sub)type of *c*) with *o.get_c_f()*.

This change permeates through to the JVM [30] where it could be realized directly by extending the `access_flags` item of the `field_info` structure and modifying the `getfield` semantics correspondingly, however, we will prefer to integrate it into the type system as described in the next section.

The first issue is simply a matter of extending the description of the field modifiers [30, §4.5] as shown in table 9.2. The second issue, changing the semantics of the `getfield` instruction [30,

Flag Name	Value
<code>ACC_PUBLIC</code>	<code>0x0001</code>
<code>ACC_PRIVATE</code>	<code>0x0002</code>
<code>ACC_PROTECTED</code>	<code>0x0004</code>
<code>:</code>	<code>:</code>
<code>ACC_RD_PUBLIC</code>	<code>0x0100</code>
<code>ACC_RD_PRIVATE</code>	<code>0x0200</code>
<code>ACC_RD_PROTECTED</code>	<code>0x0400</code>

Table 9.2: JVM read-only `access_flags` extension.

§6.4], is as easily achieved by changing the occurrence of “protected” to “*read* protected” and similarly for the associated verification constraint [30, §4.8.2].

9.3 Field Access Types

At present field access rights are checked for the `getfield` instruction both at Java verification time, for the declared “static” type [30, §4.8.2, p.138], and again at run-time, using the real “dynamic” type [30, §6.4, p.248].

Our idea is the following: if access information is integrated into the type system then

1. the bytecode verifier can check for access violations assuming that the provided access information (in the type) is correct, and
2. resolution requires equality of the used and actual type.

Thus once resolution has happened the system has checked that no access violation can happen.

Formalizing this is based on the ordering

$$\text{private} < \text{package} < \text{protected} < \text{public}$$

(where larger access rights define more accessible fields).

Assume a field is declared like

```
class c1 {
  w read r t f;
}
```

with c_1 the class hosting the field, w the “write” access modifiers, r the “read” access modifiers, t the field (value) type, and f the field name. Consider an access in a class c from within some method with code like

```
c2 x;
... x.f ...
```

(with c_2 a subclass of c_1). When we include the access modifiers in the type information this means that the field access generates the JVM instruction

$$\text{getfield}(c_1, w \text{ read } r \text{ t}, f)$$

where we note that the bytecode (as usual) contains

- the class where the field is declared: c_1 ,
- a copy of the (complete) type of the field extracted from the original definition: $w \text{ read } r \text{ t}$, and
- the field name: f .

We will express the static check for access rights of the above situation by extending the JVM type checking (verification) rules with a judgment like

$$c \vdash \text{getfield}(c_1, w \text{ read } r \text{ t}, f)$$

defined by

$$\frac{}{c \vdash \text{getfield}(c_1, w \text{ read } \text{public } t, f)}$$

$$\frac{r \geq \text{protected} \quad c \leq: c_1}{c \vdash \text{getfield}(c_1, w \text{ read } r \text{ t}, f)}$$

$$\frac{r \geq \text{package} \quad \text{same-package}(c, c_1)}{c \vdash \text{getfield}(c_1, w \text{ read } r \text{ t}, f)}$$

$$\frac{}{c \vdash \text{getfield}(c, w \text{ read } \text{private } t, f)}$$

Notice that it is the fact that the checks in table 9.1 are static that makes this possible since the verifier merely needs to be able to determine whether two classes belong to the same package and whether the current class is a subclass of the class owning the field; both can be checked with

information readily available. There are similar rules for putfield checking the *w* component of the type, of course.

All that remains is to encode the access modifiers into the JVM *FieldDescriptor* encoding [30, §4.3.2]: the existing type equality test at resolution time will ensure that the verifier has not made false assumptions. (The encoding is not difficult but outside the scope of this paper.)

9.4 Conclusion

We have outlined how access rights to fields, and specifically *read-only* access rights, can be encoded in the Java type system as implemented by a (slightly modified) Java bytecode verifier, thus eliminating all access right checks at run-time.

One very interesting further venue of research is that using “access types” could be used to implement “sticky” access rights such as “private objects” where the *value* cannot be passed out of the current method, for example.

One may comment that “static is bad because everything should be run-time configurable.” This possibility remains (using setter and getter methods) but we believe it is important to give the programmer of a class the choice of permitting (efficient) build-in static field access even for read-only fields, specifically for the variants of Java targeted at devices with limited resources [56].

Another question that one could ask is “why not for ‘setter’ methods?” This can be done but is complicated by the fact that “setter” methods usually also check the value to be stored for validity to ensure that the object is (internally) consistent. One could introduce special “validity checks” such that our getter example could be extended, for example, with a

```
class CrCardRd raises IllegalCreCaIt {
  protected read public int it
  {if (it<0) raise IllegalCreCaIt;}
}
```

with the semantics that any assignment to *it* would execute the additional “assertion” code. Such an addition may be worth considering, however, in contrast to the read-only case it complicates the Java language considerably.

Finally we remark that the above can without problems be integrated with *lightweight* bytecode verification, as used by Sun’s KVM [59], to permit static access control even in sparse resources.

Chapter 10

Conclusions

First we briefly summarize the contributions of this thesis, then we describe some related results. Finally, we give our view of possible future directions and perspectives.

10.1 Contributions

The main scientific contribution of the thesis is the specification of lightweight Java bytecode verification, along with a formal proof which shows that the type safety guarantees provided by a standard Java bytecode verifier are equally provided by the lightweight technique on an important JVM subset. The thesis demonstrates that the concept can be realized by including an implementation of a prototype, written in Java, which can be run on real “.class” files. An implementation with an observed space bound in (7.1.1a) given by $(ML + MS) \times (1 + \#LS + q)$, where q is the number of forward jump targets.

The main industrial contribution of the thesis work has been that the lightweight verification idea has been integrated into the foundation for the K-virtual machine (for resource-constrained systems), implemented by Sun Microsystems in terms of a “preverifier” that generates a “stackmap attribute” [58, 59, 28]: the “preverifier” realizes the idea of a lightweight certification stage, which establishes a certificate from the type annotations at the jump targets, and the “stackmap attribute” realizes the lightweight certificate. In order to accommodate the lightweight idea, Sun has actually *modified the Java class file standard* for the K-Virtual Machine to encompass the “stackmap” attribute. This means that at KVM today allows a method’s attribute to hold the certificate.

Finally we mention, that the formal lightweight verification specification has been mechanically verified as “correct” by Klein and Nipkow [23], as accounted for in Section 10.2. For a more exhaustive listing of the thesis contributions, we refer to Section 1.1.

10.2 Related Work

Leroy [25] gives a good overview of the problems of specifying and implementing bytecode verification, and Hartel and Moreau [20] thoroughly surveys the area of formal methods for Java safety, specifically targeted at resource-constrained devices such as Java Cards. The most interesting issues are dealt with in the literature on formalization of Java constructs in general and type safety

in particular [3, 2, 6, 10, 42, 43].

The following three works are directly related.

Sun’s StackMap attribute. In Section 8.1 we explained how this attribute can be simulated by our lightweight formalization. Here we simply point out that the KVM verifier simplifies a lightweight verification certificate by having *no label set*, but instead the frame type assignment images $\text{FTA}(\text{PP})$ stored for *all jump targets* PP . The advantage is that no delayed structures need to be activated, as all which is left to do is to check the frame types against the bytecode constraints. The drawback is that the certificate becomes *proportional in the number of code jumps*. Specifically we have that a certificate which contains a frame type at all jump targets, the space bound in (7.1.1a) collapses to a constant.

Leroy’s “On-Card” Verifier. In Section 8.2 we explained how this verifier can be simulated by our lightweight formalization. Leroy proposes a collection of (automizable) simplifications of the bytecode, prior to verification on a Java Card, which simplifies the verification tasks in several ways. In relation to our JVM code specification, he operates with the following system constraints. First, all local variable locations are assumed to be uniformly typed throughout method, that is they cannot be used to store values of other types than the given ones. Second, he assumes to know all jump targets (this set will contain our certificate labels component). Finally, it is assumed that the operand stack is empty at every such jump target. In relation to our work, this means that any certificate collapses to nothing as jumps are always between identical frame types of the form $\langle \epsilon, \text{LT} \rangle$. With respect to our verifier, the lightweight proof system will construct S and P as usual but with the peculiarity that all the values are the same, namely $\langle \epsilon, \text{LT} \rangle$. So one can optimize an implementation to not actually store these values but merely the program points in each of S and P that have this value. This optimization corresponds to the space bound of Leroy’s algorithm where only the “current” frame type is stored along with information about which program points are jump targets. With the restrictions introduced by Leroy we notice, as expected, that the space bound in (7.1.1a) collapses to a constant.

Klein and Nipkow’s Mechanically Verified Proof Klein and Nipkow [23] have formalized a restriction of lightweight bytecode verification in the Isabelle/HOL theorem prover. Their formalization is somewhat simplified by the fact that they, similarly to the “stack map attribute” of the KVM discussed above, do not compress the certificate to just the needed information, and also they do not include exceptions. The Isabelle/HOL formalization is executable thus [23] also have an executable lightweight bytecode verifier. When the certificate specifies the frame type of all jump targets then the space bound in (7.1.1a) collapses to a constant, which is indeed what the authors find from analyzing the formalizations as an algorithm (in a high-level functional language). However, the main goal of their work is merely to mechanically prove type safety and their result, verified with Isabelle/HOL, is similar to our equivalence theorem in Theorem 6.1.1. In a recent publication [62], the formalization of lightweight bytecode verification has continued beyond our JVM subset with the inclusion of object initialization.

10.3 Future Directions

We would like to extend the lightweight Java bytecode verifier to be a proper extension of the full JVM verifier, handling all of the missing issues, including packages, interfaces, initialization, all base types and multi-dimensional arrays, static components, and inner classes.

Another interesting perspective is to investigate how to replace existing runtime checks by static checks by representing more information in the type system. An example of this is the access protection checks discussed in Chapter 9, but many more are interesting, including array and jump table bound checks, overflow checks, and null pointer avoidance.

Appendix A

cksum () Example Details

This appendix gives the details of the `cksum()` method example introduced in Example 3.2.14 (page 45) and used throughout the thesis.

A.1 Standard Verification Proof

This section gives the details of the proof for Proposition 4.5.1. We assume all the definitions of that proof except we shall omit the ^{ck}-superscripts and impose a structure on the code $C = \text{iload}[1] \cdot CS_0$ where $CS_0 = \text{invokevirtual}[1] \cdot CS_2, \dots, CS_{57} = \text{goto}[-39]$ (one can read CS_{PP} as “the code following the instruction at PP”). Throughout we use \Rightarrow and \rightarrow for their ‘tsafe’-annotated counterparts in Chapter 4, and we have abbreviated the subsets of PPS with “...”. Finally the following convention is used for inference rules.

Notation A.1.1 (proof trees). Because the unfoldings are rather extensive we shall write

$$\frac{\begin{array}{l} \boxed{\text{premise}_1} \text{ (Axiom)} \\ \vdots \\ \boxed{\text{premise}_k} \text{ (Axiom)} \end{array}}{\text{conclusion} \text{ (Rule)}}$$

for the inference proof fragment

$$\frac{\boxed{\text{premise}_1} \text{ (Axiom)} \quad \dots \quad \boxed{\text{premise}_k} \text{ (Axiom)}}{\text{conclusion} \text{ (Rule)}}$$

A boxed premise is a reference to another definition where the proof of the premise(s) is given.

$$(A.1.2) \quad \frac{\begin{array}{l} \boxed{\Omega \vdash 0 : \text{aload}[1] \rightarrow 2, \text{FTA}(2)} \text{ (4.3.8a)} \\ \boxed{CH \vdash \text{FTA}(2) \sqsubseteq \text{FTA}(2)} \text{ (4.1.28a)} \\ \boxed{A.1.3} \end{array}}{\Omega \vdash 0, \text{aload}[1] \cdot \text{invokevirtual}[1] \cdot CS_2, \emptyset \Rightarrow \text{PPS}} \text{ (4.2.8b)}$$

$$\begin{array}{l}
\text{(A.1.3)} \quad \left[\begin{array}{l}
\overline{\text{CH} \vdash \text{FTA}(10) \sqsubseteq \langle \text{UnsetCrCard}, \text{LT}_{10} \rangle} \quad (4.1.28a) \\
\overline{\text{CH} \vdash \text{FTA}(10) \sqsubseteq \langle \text{UnsetCrCard}, \text{LT}_{10} \rangle} \quad (4.1.28a) \\
\left[\overline{\text{CH} \vdash \text{FTA}(10) \sqsubseteq \langle \text{UnsetCrCard}, \text{LT}_{10} \rangle} \quad (4.1.28a) \right. \\
\left. \left[\overline{\Theta \vdash 2, \epsilon, \epsilon} \quad (4.4.12c) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \epsilon, \text{ET}} \quad (4.4.16a) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{UnsetCrCard}, \epsilon} \quad (4.4.12b) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{UnsetCrCard}, \text{ET}} \quad (4.4.14a) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Abort} \cdot \text{UnsetCrCard}, \epsilon} \quad (4.4.12b) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Abort} \cdot \text{UnsetCrCard}, \text{ET}} \quad (4.4.16a) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Exception} \cdot \text{Abort} \cdot \text{UnsetCrCard}, \epsilon} \quad (4.4.12b) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Exception} \cdot \text{Abort} \cdot \text{UnsetCrCard}, \text{ET}} \quad (4.4.16a) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Throwable} \cdot \text{Exception} \cdot \text{Abort} \cdot \text{UnsetCrCard}, \epsilon} \quad (4.4.12b) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Throwable} \cdot \text{Exception} \cdot \text{Abort} \cdot \text{UnsetCrCard}, \text{ET}} \quad (4.4.16a) \right. \right. \\
\left. \left. \overline{\Theta \vdash 2, \text{Throwable} \cdot \text{Exception} \cdot \text{Abort} \cdot \text{UnsetCrCard}, \text{ET}} \quad (4.3.17a) \right. \right. \\
\overline{\Omega \vdash 2 : \text{invokevirtual}[1] \rightarrow 5, \text{FTA}(5)} \\
\overline{\text{CH} \vdash \text{FTA}(5) \sqsubseteq \text{FTA}(5)} \quad (4.1.28a) \\
\boxed{\text{A.1.4}} \\
\overline{\Omega \vdash 2, \text{invokevirtual}[1] \cdot \text{istore}[2] \cdot \text{CS}_5, \{0\} \Rightarrow \text{PPS}} \quad (4.2.8b)
\end{array}
\right.
\end{array}$$

where $\Theta = \langle \Omega, \text{LT}_2, \text{false} \rangle$ with the contained list of possible exceptions obtained from $\{E \mid E \leq_{\text{CH}} \text{Throwable}\}$, and with $\text{FTA}(10) = \langle \text{ST}_{10}, \text{LT}_{10} \rangle$.

$$\begin{array}{l}
\text{(A.1.4)} \quad \left[\begin{array}{l}
\overline{\Omega \vdash 5 : \text{istore}[2] \rightarrow 7, \text{FTA}(7)} \quad (4.3.8a) \\
\overline{\text{CH} \vdash \text{FTA}(7) \sqsubseteq \text{FTA}(7)} \quad (4.1.28a) \\
\boxed{\text{A.1.5}} \\
\overline{\Omega \vdash 5, \text{istore}[2] \cdot \text{goto}[+8] \cdot \text{CS}_7, \{0, 2\} \Rightarrow \text{PPS}} \quad (4.2.8b)
\end{array}
\right.
\end{array}$$

$$\begin{array}{l}
\text{(A.1.5)} \quad \left[\begin{array}{l}
\overline{\text{CH} \vdash \text{FTA}(15) \sqsubseteq \text{FTA}(7)} \quad (4.1.28a) \\
\overline{\Omega \vdash 7 : \text{goto}[+8] \rightarrow 10, \top} \quad (4.3.22a) \\
\overline{\text{CH} \vdash \text{FTA}(10) \sqsubseteq \top} \quad (4.1.28d) \\
\boxed{\text{A.1.6}} \\
\overline{\Omega \vdash 7, \text{goto}[+8] \cdot \text{pop} \cdot \text{CS}_{10}, \{0, 2, 5\} \Rightarrow \text{PPS}} \quad (4.2.8b)
\end{array}
\right.
\end{array}$$

$$(A.1.6) \quad \frac{\begin{array}{l} \frac{}{\Omega \vdash 10 : \text{pop} \rightarrow 11, \text{FTA}(11)} \text{ (4.3.6a)} \\ \frac{}{\text{CH} \vdash \text{FTA}(11) \sqsubseteq \text{FTA}(11)} \text{ (4.1.28a)} \\ \boxed{\text{A.1.7}} \end{array}}{\Omega \vdash 10, \text{pop} \cdot \text{new}[2] \cdot \text{CS}_{11}, \{0, 2, 5, 7\} \Rightarrow \text{PPS}} \text{ (4.2.8b)}$$

$$(A.1.7) \quad \frac{\begin{array}{l} \frac{}{\langle \Omega, \text{LT}_{11}, \text{false} \rangle \vdash 11, \epsilon, \text{ET}} \text{ (4.4.12b)} \\ \frac{}{\Omega \vdash 11 : \text{new}[2] \rightarrow 14, \text{FTA}(14)} \text{ (4.3.13a)} \\ \frac{}{\text{CH} \vdash \text{FTA}(14) \sqsubseteq \text{FTA}(14)} \text{ (4.1.28a)} \\ \boxed{\text{A.1.8}} \end{array}}{\Omega \vdash 11, \text{new}[2] \cdot \text{athrow} \cdot \text{CS}_{14}, \{0, 2, 5, 7, 10\} \Rightarrow \text{PPS}} \text{ (4.2.8b)}$$

where LT_{11} is defined by $\text{FTA}(11) = \langle -, \text{LT}_{11} \rangle$.

$$(A.1.8) \quad \frac{\begin{array}{l} \frac{}{\text{CH} \vdash \text{Abort}, \text{EA}} \text{ (4.4.20b)} \\ \frac{}{\text{CH} \vdash \text{true}, \text{Abort}, \text{EA}} \text{ (4.4.19b)} \\ \frac{}{\langle \Omega, \text{LT}_{14}, \text{true} \rangle \vdash 14, \epsilon, \text{ET}} \text{ (4.4.12c)} \\ \frac{}{\langle \Omega, \text{LT}_{14}, \text{true} \rangle \vdash 14, \text{Abort}, \epsilon} \text{ (4.4.12b)} \\ \frac{}{\langle \Omega, \text{LT}_{14}, \text{true} \rangle \vdash 14, \text{Abort}, \text{ET}} \text{ (4.4.12d)} \\ \frac{}{\Omega \vdash 14 : \text{athrow} \rightarrow 15, \top} \text{ (4.3.24b)} \\ \frac{}{\text{CH} \vdash \text{FTA}(15) \sqsubseteq \top} \text{ (4.1.28d)} \\ \boxed{\text{A.1.9}} \end{array}}{\Omega \vdash 14, \text{athrow} \cdot \text{ldc.w}[3] \cdot \text{CS}_{15}, \{0, 2, \dots, 11\} \Rightarrow \text{PPS}} \text{ (4.2.8b)}$$

where LT_{14} is defined by $\text{FTA}(14) = \langle -, \text{LT}_{14} \rangle$.

$$(A.1.9) \quad \frac{\begin{array}{l} \frac{}{\langle \Omega, \text{LT}_{15}, \text{false} \rangle \vdash 15, \epsilon, \text{ET}} \text{ (4.4.12b)} \\ \frac{}{\Omega \vdash 15 : \text{ldc.w}[3] \rightarrow 18, \text{FTA}(18)} \text{ (4.3.13a)} \\ \frac{}{\text{CH} \vdash \text{FTA}(18) \sqsubseteq \text{FTA}(18)} \text{ (4.1.28a)} \\ \boxed{\text{A.1.10}} \end{array}}{\Omega \vdash 15, \text{ldc.w}[3] \cdot \text{istore}[3] \cdot \text{CS}_{18}, \{0, 2, \dots, 14\} \Rightarrow \text{PPS}} \text{ (4.2.8b)}$$

where LT_{15} is defined by $\text{FTA}(15) = \langle -, \text{LT}_{15} \rangle$.

$$\begin{array}{l}
\text{(A.1.10)} \quad \left[\begin{array}{l} \overline{\Omega \vdash 18 : \text{istore}[3] \rightarrow 20, \text{FTA}(20)}^{(4.3.8a)} \\ \overline{\text{CH} \vdash \text{FTA}(20) \sqsubseteq \text{FTA}(20)}^{(4.1.28a)} \\ \boxed{\text{A.1.11}} \end{array} \right] \\
\overline{\Omega \vdash 18, \text{istore}[3] \cdot \text{iload}[2] \cdot \text{CS}_{20, \{0, 2, \dots, 15\}} \Rightarrow \text{PPS}}^{(4.2.8b)}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.11)} \quad \left[\begin{array}{l} \overline{\Omega \vdash 20 : \text{iload}[2] \rightarrow 22, \text{FTA}(22)}^{(4.3.8a)} \\ \overline{\text{CH} \vdash \text{FTA}(22) \sqsubseteq \text{FTA}(22)}^{(4.1.28a)} \\ \boxed{\text{A.1.12}} \end{array} \right] \\
\overline{\Omega \vdash 20, \text{iload}[2] \cdot \text{iload}[3] \cdot \text{CS}_{22, \{0, 2, \dots, 18\}} \Rightarrow \text{PPS}}^{(4.2.8b)}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.12)} \quad \left[\begin{array}{l} \overline{\Omega \vdash 22 : \text{iload}[3] \rightarrow 24, \text{FTA}(24)}^{(4.3.8a)} \\ \overline{\text{CH} \vdash \text{FTA}(24) \sqsubseteq \text{FTA}(24)}^{(4.1.28a)} \\ \boxed{\text{A.1.13}} \end{array} \right] \\
\overline{\Omega \vdash 22, \text{iload}[3] \cdot \text{isub} \cdot \text{CS}_{24, \{0, 2, \dots, 20\}} \Rightarrow \text{PPS}}^{(4.2.8b)}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.13)} \quad \left[\begin{array}{l} \overline{\Omega \vdash 24 : \text{isub} \rightarrow 25, \text{FTA}(25)}^{(4.3.6a)} \\ \overline{\text{CH} \vdash \text{FTA}(25) \sqsubseteq \text{FTA}(25)}^{(4.1.28a)} \\ \boxed{\text{A.1.14}} \end{array} \right] \\
\overline{\Omega \vdash 24, \text{isub} \cdot \text{istore}[4] \cdot \text{CS}_{25, \{0, 2, \dots, 22\}} \Rightarrow \text{PPS}}^{(4.2.8b)}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.14)} \quad \left[\begin{array}{l} \overline{\Omega \vdash 25 : \text{istore}[4] \rightarrow 27, \text{FTA}(27)}^{(4.3.8a)} \\ \overline{\text{CH} \vdash \text{FTA}(27) \sqsubseteq \text{FTA}(27)}^{(4.1.28a)} \\ \boxed{\text{A.1.15}} \end{array} \right] \\
\overline{\Omega \vdash 25, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{27, \{0, 2, \dots, 24\}} \Rightarrow \text{PPS}}^{(4.2.8b)}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.15)} \quad \left[\begin{array}{l} \overline{\Omega \vdash 27 : \text{iload}[3] \rightarrow 29, \text{FTA}(29)}^{(4.3.8a)} \\ \overline{\text{CH} \vdash \text{FTA}(29) \sqsubseteq \text{FTA}(29)}^{(4.1.28a)} \\ \boxed{\text{A.1.16}} \end{array} \right] \\
\overline{\Omega \vdash 27, \text{iload}[3] \cdot \text{ifle}[+10] \cdot \text{CS}_{29, \{0, 2, \dots, 25\}} \Rightarrow \text{PPS}}^{(4.2.8b)}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.16)} \\
\hline
\begin{array}{l}
\overline{\text{CH} \vdash \text{FTA}(39) \sqsubseteq \text{FTA}(32)} \quad (4.1.28a) \\
\overline{\Omega \vdash 29 : \text{ifl}[+10] \rightarrow 32, \text{FTA}(32)} \quad (5.3.8b) \\
\overline{\text{CH} \vdash \text{FTA}(32) \sqsubseteq \text{FTA}(32)} \quad (4.1.28a) \\
\boxed{\text{A.1.17}}
\end{array} \\
\hline
\Omega \vdash 29, \text{ifl}[+10] \cdot \text{iload}[4] \cdot \text{CS}_{32}, \{0, 2, \dots, 27\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{array}$$

$$\begin{array}{l}
\text{(A.1.17)} \\
\hline
\begin{array}{l}
\overline{\Omega \vdash 32 : \text{iload}[4] \rightarrow 34, \text{FTA}(34)} \quad (4.3.8a) \\
\overline{\text{CH} \vdash \text{FTA}(34) \sqsubseteq \text{FTA}(34)} \quad (4.1.28a) \\
\boxed{\text{A.1.18}}
\end{array} \\
\hline
\Omega \vdash 32, \text{iload}[4] \cdot \text{istore}[2] \cdot \text{CS}_{34}, \{0, 2, \dots, 29\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{array}$$

$$\begin{array}{l}
\text{(A.1.18)} \\
\hline
\begin{array}{l}
\overline{\Omega \vdash 34 : \text{istore}[2] \rightarrow 36, \text{FTA}(36)} \quad (4.3.8a) \\
\overline{\text{CH} \vdash \text{FTA}(36) \sqsubseteq \text{FTA}(36)} \quad (4.1.28a) \\
\boxed{\text{A.1.19}}
\end{array} \\
\hline
\Omega \vdash 34, \text{istore}[2] \cdot \text{goto}[-16] \cdot \text{CS}_{36}, \{0, 2, \dots, 32\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{array}$$

$$\begin{array}{l}
\text{(A.1.19)} \\
\hline
\begin{array}{l}
\overline{\text{CH} \vdash \text{FTA}(20) \sqsubseteq \text{FTA}(36)} \quad (4.1.28a) \\
\overline{\Omega \vdash 36 : \text{goto}[-16] \rightarrow 39, \top} \quad (4.3.22a) \\
\overline{\text{CH} \vdash \text{FTA}(39) \sqsubseteq \top} \quad (4.1.28d) \\
\boxed{\text{A.1.20}}
\end{array} \\
\hline
\Omega \vdash 36, \text{goto}[-16] \cdot \text{iload}[4] \cdot \text{CS}_{39}, \{0, 2, \dots, 34\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{array}$$

$$\begin{array}{l}
\text{(A.1.20)} \\
\hline
\begin{array}{l}
\overline{\Omega \vdash 39 : \text{iload}[4] \rightarrow 41, \text{FTA}(41)} \quad (4.3.8a) \\
\overline{\text{CH} \vdash \text{FTA}(41) \sqsubseteq \text{FTA}(41)} \quad (4.1.28a) \\
\boxed{\text{A.1.21}}
\end{array} \\
\hline
\Omega \vdash 39, \text{iload}[4] \cdot \text{ifne}[+6] \cdot \text{CS}_{41}, \{0, 2, \dots, 36\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{array}$$

$$\begin{array}{l}
\text{(A.1.21)} \\
\hline
\begin{array}{l}
\overline{\text{CH} \vdash \text{FTA}(47) \sqsubseteq \text{FTA}(44)} \quad (4.1.28a) \\
\overline{\Omega \vdash 41 : \text{ifne}[+6] \rightarrow 44, \text{FTA}(44)} \quad (4.3.20a) \\
\overline{\text{CH} \vdash \text{FTA}(44) \sqsubseteq \text{FTA}(44)} \quad (4.1.28a) \\
\boxed{\text{A.1.22}}
\end{array} \\
\hline
\Omega \vdash 41, \text{ifne}[+6] \cdot \text{iload}[2] \cdot \text{CS}_{44}, \{0, 2, \dots, 39\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{array}$$

$$\begin{array}{l}
\text{(A.1.22)} \quad \frac{\frac{\frac{}{\Omega \vdash 44 : \text{iload}[2] \rightarrow 46, \text{FTA}(46)} \quad (4.3.8a)}{\frac{}{\text{CH} \vdash \text{FTA}(46) \sqsubseteq \text{FTA}(46)} \quad (4.1.28a)}}{\boxed{\text{A.1.23}}} \quad (4.2.8b)}{\Omega \vdash 44, \text{iload}[2] \cdot \text{ireturn} \cdot \text{CS}_{46}, \{0, 2, \dots, 41\} \Rightarrow \text{PPS}}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.23)} \quad \frac{\frac{\frac{}{\Omega \vdash 46 : \text{ireturn} \rightarrow 47, \top} \quad (4.3.25)}{\frac{}{\text{CH} \vdash \text{FTA}(47) \sqsubseteq \top} \quad (4.1.28d)}}{\boxed{\text{A.1.24}}} \quad (4.2.8b)}{\Omega \vdash 46, \text{ireturn} \cdot \text{iload}[2] \cdot \text{CS}_{47}, \{0, 2, \dots, 44\} \Rightarrow \text{PPS}}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.24)} \quad \frac{\frac{\frac{}{\Omega \vdash 47 : \text{iload}[2] \rightarrow 49, \text{FTA}(49)} \quad (4.3.8a)}{\frac{}{\text{CH} \vdash \text{FTA}(49) \sqsubseteq \text{FTA}(49)} \quad (4.1.28a)}}{\boxed{\text{A.1.25}}} \quad (4.2.8b)}{\Omega \vdash 47, \text{iload}[2] \cdot \text{istore}[4] \cdot \text{CS}_{49}, \{0, 2, \dots, 46\} \Rightarrow \text{PPS}}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.25)} \quad \frac{\frac{\frac{}{\Omega \vdash 49 : \text{istore}[4] \rightarrow 51, \text{FTA}(51)} \quad (4.3.8a)}{\frac{}{\text{CH} \vdash \text{FTA}(51) \sqsubseteq \text{FTA}(51)} \quad (4.1.28a)}}{\boxed{\text{A.1.26}}} \quad (4.2.8b)}{\Omega \vdash 49, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{51}, \{0, 2, \dots, 47\} \Rightarrow \text{PPS}}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.26)} \quad \frac{\frac{\frac{}{\Omega \vdash 51 : \text{iload}[3] \rightarrow 53, \text{FTA}(53)} \quad (4.3.8a)}{\frac{}{\text{CH} \vdash \text{FTA}(53) \sqsubseteq \text{FTA}(53)} \quad (4.1.28a)}}{\boxed{\text{A.1.27}}} \quad (4.2.8b)}{\Omega \vdash 51, \text{iload}[3] \cdot \text{istore}[2] \cdot \text{CS}_{53}, \{0, 2, \dots, 49\} \Rightarrow \text{PPS}}
\end{array}$$

$$\begin{array}{l}
\text{(A.1.27)} \quad \frac{\frac{\frac{}{\Omega \vdash 53 : \text{istore}[2] \rightarrow 55, \text{FTA}(55)} \quad (4.3.8a)}{\frac{}{\text{CH} \vdash \text{FTA}(55) \sqsubseteq \text{FTA}(55)} \quad (4.1.28a)}}{\boxed{\text{A.1.28}}} \quad (4.2.8b)}{\Omega \vdash 53, \text{istore}[2] \cdot \text{iload}[4] \cdot \text{CS}_{55}, \{0, 2, \dots, 51\} \Rightarrow \text{PPS}}
\end{array}$$

$$(A.1.28) \quad \frac{\frac{\frac{}{\Omega \vdash 55 : \text{iload}[4] \rightarrow 57, \text{FTA}(57)} \quad (4.3.8a)}{\text{CH} \vdash \text{FTA}(57) \sqsubseteq \text{FTA}(57)} \quad (4.1.28a)}{\boxed{\text{A.1.29}}} \quad (4.2.8b)}{\Omega \vdash 55, \text{iload}[4] \cdot \text{istore}[3] \cdot \text{CS}_{57}, \{0, 2, \dots, 53\} \Rightarrow \text{PPS}}$$

$$(A.1.29) \quad \frac{\frac{\frac{}{\Omega \vdash 57 : \text{istore}[3] \rightarrow 59, \text{FTA}(59)} \quad (4.3.8a)}{\text{CH} \vdash \text{FTA}(59) \sqsubseteq \text{FTA}(59)} \quad (4.1.28a)}{\boxed{\text{A.1.30}}} \quad (4.2.8b)}{\Omega \vdash 57, \text{istore}[3] \cdot \text{goto}[-39] \cdot \epsilon, \{0, 2, \dots, 55\} \Rightarrow \text{PPS}}$$

$$(A.1.30) \quad \frac{\frac{}{\Omega \vdash 59 : \text{goto}[-39] \rightarrow 62, \overline{\top}} \quad (4.3.22a)}{\Omega \vdash 59, \text{goto}[-39], \{0, 2, \dots, 55, 57\} \Rightarrow \text{PPS}} \quad (4.2.8a)$$

A.2 Lightweight Verification Proof

This section gives the details of the proof for the lightweight bytecode verification Proposition 5.5.1 (on page 120). We assume the same definitions as in that proof and allow shorthands as in the previous section except in this section we allow \Rightarrow and \rightarrow for their 'ltsafe'-annotated counterparts of Chapter 5. Finally we permit reference to FTA of Figure 4.5 since it provides just the right the frame type assignments in most cases.

$$(A.2.1) \quad \frac{\frac{\frac{}{\Omega_{\text{light}} \vdash \langle 0, \text{FTA}(0) \rangle, \text{aload}[1], \langle P_0, S_0 \rangle \rightarrow \langle 2, \text{FTA}(2) \rangle, P_0} \quad (5.3.3a)}{\boxed{\text{A.2.2}}} \quad (5.2.6b)}{\Omega_{\text{light}} \vdash \langle 0, \text{FTA}(0) \rangle, \text{aload}[1] \cdot \text{invokevirtual}[1] \cdot \text{CS}_2, \emptyset, \langle P_0, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}}$$

where $P_{10} = P_7 \sqcap \{10 \mapsto \top\}$.

(A.2.6)

$$\frac{\frac{\frac{\frac{}{\langle \Omega_{\text{light}}, \text{LT}_{11}, \text{false} \rangle \vdash 11, \epsilon, \text{ET}}{\langle \Omega_{\text{light}}, \text{LT}_{11}, \text{false} \rangle \vdash 11, \epsilon, \text{ET}} \quad (5.4.5b)}{\Omega_{\text{light}} \vdash \langle 11, \text{FTA}(11) \rangle, \text{new}[2], \langle P_{10}, S_0 \rangle \rightarrow \langle 14, \text{FTA}(14) \rangle, P_{10}} \quad (4.3.13a)}{\boxed{\text{A.2.7}}}}{\Omega_{\text{light}} \vdash \langle 11, \text{FTA}(11) \rangle, \text{new}[2] \cdot \text{athrow} \cdot \text{CS}_{14}, \{0, \dots, 10\}, \langle P_{10}, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

where LT_{11} is defined by $\text{FTA}(11) = \langle -, \text{LT}_{11} \rangle$.

(A.2.7)

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{}{\text{CH} \vdash \text{Abort}, \text{EA}}{\text{CH} \vdash \text{true}, \text{Abort}, \text{EA}} \quad (4.4.19b)}{\frac{}{\langle \Omega_{\text{light}}, \text{LT}_{14}, \text{true} \rangle \vdash 14, \epsilon, \text{ET}, \langle P_{10}, S_0 \rangle \rightarrow P_{10}} \quad (5.4.5c)}{\langle \Omega_{\text{light}}, \text{LT}_{14}, \text{true} \rangle \vdash 14, \text{Abort}, \epsilon, \langle P_{10}, S_0 \rangle \rightarrow P_{10}} \quad (5.4.5b)}{\langle \Omega_{\text{light}}, \text{LT}_{14}, \text{true} \rangle \vdash 14, \text{Abort}, \text{ET}, \langle P_{10}, S_0 \rangle \rightarrow P_{10}} \quad (5.4.5d)}{\Omega_{\text{light}} \vdash \langle 14, \text{FTA}(14) \rangle, \text{athrow}, \langle P_{10}, S_0 \rangle \rightarrow \langle 15, \top \rangle, P_{10}} \quad (5.3.10b)}{\boxed{\text{A.2.8}}}}{\Omega_{\text{light}} \vdash \langle 14, \text{FTA}(14) \rangle, \text{athrow} \cdot \text{ldc_w}[3] \cdot \text{CS}_{15}, \{0, \dots, 11\}, \langle P_{10}, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

where LT_{14} is defined by $\text{FTA}(14) = \langle -, \text{LT}_{14} \rangle$.

(A.2.8)

$$\frac{\frac{\frac{\frac{}{\langle \Omega_{\text{light}}, \text{LT}_{15}, \text{false} \rangle \vdash 15, \epsilon, \text{ET}}{\langle \Omega_{\text{light}}, \text{LT}_{15}, \text{false} \rangle \vdash 15, \epsilon, \text{ET}} \quad (5.4.5b)}{\Omega_{\text{light}} \vdash \langle 15, \text{FTA}(15) \rangle, \text{ldc_w}[3], \langle P_{10}, S_0 \rangle \rightarrow \langle 18, \text{FTA}(18) \rangle, P_0} \quad (5.3.5a)}{\boxed{\text{A.2.9}}}}{\Omega_{\text{light}} \vdash \langle 15, \text{FTA}(15) \rangle, \text{ldc_w}[3] \cdot \text{istore}[3] \cdot \text{CS}_{18}, \{0, \dots, 14\}, \langle P_0, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

where LT_{15} is defined by $\text{FTA}(15) = \langle -, \text{LT}_{15} \rangle$.

(A.2.9)

$$\frac{\frac{\frac{}{\Omega_{\text{light}} \vdash \langle 18, \text{FTA}(18) \rangle, \text{istore}[3], \langle P_0, S_0 \rangle \rightarrow \langle 20, \text{FTA}(20) \rangle, P_0} \quad (5.3.3a)}{\boxed{\text{A.2.10}}}}{\Omega_{\text{light}} \vdash \langle 18, \text{FTA}(18) \rangle, \text{istore}[3] \cdot \text{iload}[2] \cdot \text{CS}_{20}, \{0, \dots, 15\}, \langle P_0, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.10)

$$\frac{\boxed{\text{A.2.11}} \quad \overline{\Omega_{\text{light}} \vdash \langle 20, \text{FTA}(20) \rangle, \text{iload}[2], \langle P_0, S_{20} \rangle \rightarrow \langle 22, \text{FTA}(22) \rangle, P_0}^{(5.3.3a)}}{\Omega_{\text{light}} \vdash \langle 20, \text{FTA}(20) \rangle, \text{iload}[2] \cdot \text{iload}[3] \cdot \text{CS}_{22, \{0, \dots, 18\}}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.11)

$$\frac{\boxed{\text{A.2.12}} \quad \overline{\Omega_{\text{light}} \vdash \langle 22, \text{FTA}(22) \rangle, \text{iload}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 24, \text{FTA}(24) \rangle, P_0}^{(5.3.3a)}}{\Omega_{\text{light}} \vdash \langle 22, \text{FTA}(22) \rangle, \text{iload}[3] \cdot \text{isub} \cdot \text{CS}_{24, \{0, \dots, 20\}}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.12)

$$\frac{\boxed{\text{A.2.13}} \quad \overline{\Omega_{\text{light}} \vdash \langle 24, \text{FTA}(24) \rangle, \text{isub}, \langle P_0, S_{20} \rangle \rightarrow \langle 25, \text{FTA}(25) \rangle, P_0}^{(5.3.2a)}}{\Omega_{\text{light}} \vdash \langle 24, \text{FTA}(24) \rangle, \text{isub} \cdot \text{istore}[4] \cdot \text{CS}_{25, \{0, \dots, 22\}}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.13)

$$\frac{\boxed{\text{A.2.14}} \quad \overline{\Omega_{\text{light}} \vdash \langle 25, \text{FTA}(25) \rangle, \text{istore}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 27, \text{FTA}(27) \rangle, P_0}^{(5.3.3a)}}{\Omega_{\text{light}} \vdash \langle 25, \text{FTA}(25) \rangle, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{27, \{0, \dots, 24\}}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.14)

$$\frac{\boxed{\text{A.2.15}} \quad \overline{\Omega_{\text{light}} \vdash \langle 27, \text{FTA}(27) \rangle, \text{iload}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 29, \text{FTA}(29) \rangle, P_0}^{(5.3.3a)}}{\Omega_{\text{light}} \vdash \langle 27, \text{FTA}(27) \rangle, \text{iload}[3] \cdot \text{ifle}[+10] \cdot \text{CS}_{29, \{0, \dots, 25\}}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.15)

$$\frac{\boxed{\text{A.2.16}} \quad \overline{\Omega_{\text{light}} \vdash \langle 29, \text{FTA}(29) \rangle, \text{ifle}[+10], \langle P_0, S_{20} \rangle \rightarrow \langle 32, \text{FTA}(32) \rangle, P_{29}}^{(5.3.8b)}}{\Omega_{\text{light}} \vdash \langle 29, \text{FTA}(29) \rangle, \text{ifle}[+10] \cdot \text{iload}[4] \cdot \text{CS}_{32, \{0, \dots, 27\}}, \langle P_{29}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

where $P_{29} = P_0 \sqcap \{39 \mapsto \text{FTA}(39)\}$.

(A.2.16)

$$\frac{\boxed{\text{A.2.17}} \quad \overline{\Omega_{\text{light}} \vdash \langle 32, \text{FTA}(32) \rangle, \text{iload}[4], \langle P_{29}, S_{20} \rangle \rightarrow \langle 34, \text{FTA}(34) \rangle, P_{29}}^{(5.3.3a)}}{\Omega_{\text{light}} \vdash \langle 32, \text{FTA}(32) \rangle, \text{iload}[4] \cdot \text{istore}[2] \cdot \text{CS}_{34, \{0, \dots, 29\}}, \langle P_{29}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.17)

$$\frac{\boxed{\text{A.2.18}} \quad \overline{\Omega_{\text{light}} \vdash \langle 34, \text{FTA}(34) \rangle, \text{istore}[2], \langle P_{29}, S_{20} \rangle \rightarrow \langle 36, \text{FTA}(36) \rangle, P_{29}} \quad (5.3.3a)}{\Omega_{\text{light}} \vdash \langle 34, \text{FTA}(34) \rangle, \text{istore}[2] \cdot \text{goto}[-16] \cdot \text{CS}_{36}, \{0, \dots, 32\}, \langle P_{29}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.18)

$$\frac{\boxed{\text{A.2.19}} \quad \overline{\Omega_{\text{light}} \vdash \langle 36, \text{FTA}(36) \rangle, \text{goto}[-16], \langle P_{29}, S_{20} \rangle \rightarrow \langle 39, \top \rangle, P_{29}} \quad (5.3.9a)}{\Omega_{\text{light}} \vdash \langle 36, \text{FTA}(36) \rangle, \text{goto}[-16] \cdot \text{iload}[4] \cdot \text{CS}_{39}, \{0, \dots, 34\}, \langle P_{29}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.19)

$$\frac{\boxed{\text{A.2.20}} \quad \overline{\Omega_{\text{light}} \vdash \langle 39, \text{FTA}(39) \rangle, \text{iload}[4], \langle P_{29}, S_{20} \rangle \rightarrow \langle 41, \text{FTA}(41) \rangle, P_0} \quad (5.3.3a)}{\Omega_{\text{light}} \vdash \langle 39, \text{FTA}(39) \rangle, \text{iload}[4] \cdot \text{ifne}[+6] \cdot \text{CS}_{41}, \{0, \dots, 36\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.20)

$$\frac{\boxed{\text{A.2.21}} \quad \overline{\Omega_{\text{light}} \vdash \langle 41, \text{FTA}(41) \rangle, \text{ifne}[+6], \langle P_0, S_{20} \rangle \rightarrow \langle 44, \text{FTA}(44) \rangle, P_{41}} \quad (5.3.8b)}{\Omega_{\text{light}} \vdash \langle 41, \text{FTA}(41) \rangle, \text{ifne}[+6] \cdot \text{iload}[2] \cdot \text{CS}_{44}, \{0, \dots, 39\}, \langle P_{41}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

where $P_{41} = P_0 \sqcap \{47 \mapsto \text{FTA}(47)\}$ here and below.

(A.2.21)

$$\frac{\boxed{\text{A.2.22}} \quad \overline{\Omega_{\text{light}} \vdash \langle 44, \text{FTA}(44) \rangle, \text{iload}[2], \langle P_{41}, S_{20} \rangle \rightarrow \langle 46, \text{FTA}(46) \rangle, P_{41}} \quad (5.3.3a)}{\Omega_{\text{light}} \vdash \langle 44, \text{FTA}(44) \rangle, \text{iload}[2] \cdot \text{ireturn} \cdot \text{CS}_{46}, \{0, \dots, 41\}, \langle P_{41}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.22)

$$\frac{\boxed{\text{A.2.23}} \quad \overline{\Omega_{\text{light}} \vdash \langle 46, \text{FTA}(46) \rangle, \text{ireturn}, \langle P_{41}, S_{20} \rangle \rightarrow \langle 47, \top \rangle, P_{41}} \quad (5.3.11a)}{\Omega_{\text{light}} \vdash \langle 46, \text{FTA}(46) \rangle, \text{ireturn} \cdot \text{iload}[2] \cdot \text{CS}_{47}, \{0, \dots, 44\}, \langle P_{41}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.23)

$$\frac{\boxed{\text{A.2.24}} \quad \overline{\Omega_{\text{light}} \vdash \langle 47, \text{FTA}(47) \rangle, \text{iload}[2], \langle P_{41}, S_{20} \rangle \rightarrow \langle 49, \text{FTA}(49) \rangle, P_0} \quad (5.3.3a)}{\Omega_{\text{light}} \vdash \langle 47, \text{FTA}(47) \rangle, \text{iload}[2] \cdot \text{istore}[4] \cdot \text{CS}_{49}, \{0, \dots, 46\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle} \quad (5.2.6b)$$

(A.2.24)

$$\frac{\overline{\Omega_{\text{light}} \vdash \langle 49, \text{FTA}(49) \rangle, \text{istore}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 51, \text{FTA}(51) \rangle, P_0}^{(5.3.3a)}}{\boxed{\text{A.2.25}}} \frac{}{\Omega_{\text{light}} \vdash \langle 49, \text{FTA}(49) \rangle, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{51}, \{0, \dots, 47\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.25)

$$\frac{\overline{\Omega_{\text{light}} \vdash \langle 51, \text{FTA}(51) \rangle, \text{iload}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 53, \text{FTA}(53) \rangle, P_0}^{(5.3.3a)}}{\boxed{\text{A.2.26}}} \frac{}{\Omega_{\text{light}} \vdash \langle 51, \text{FTA}(51) \rangle, \text{iload}[3] \cdot \text{istore}[2] \cdot \text{CS}_{53}, \{0, \dots, 49\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.26)

$$\frac{\overline{\Omega_{\text{light}} \vdash \langle 53, \text{FTA}(53) \rangle, \text{istore}[2], \langle P_0, S_{20} \rangle \rightarrow \langle 55, \text{FTA}(55) \rangle, P_0}^{(5.3.3a)}}{\boxed{\text{A.2.27}}} \frac{}{\Omega_{\text{light}} \vdash \langle 53, \text{FTA}(53) \rangle, \text{istore}[2] \cdot \text{iload}[4] \cdot \text{CS}_{55}, \{0, \dots, 51\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.27)

$$\frac{\overline{\Omega_{\text{light}} \vdash \langle 55, \text{FTA}(55) \rangle, \text{iload}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 57, \text{FTA}(57) \rangle, P_0}^{(5.3.3a)}}{\boxed{\text{A.2.28}}} \frac{}{\Omega_{\text{light}} \vdash \langle 55, \text{FTA}(55) \rangle, \text{iload}[4] \cdot \text{istore}[3] \cdot \text{CS}_{57}, \{0, \dots, 53\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

(A.2.28)

$$\frac{\overline{\Omega_{\text{light}} \vdash \langle 57, \text{FTA}(57) \rangle, \text{istore}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 59, \text{FTA}(59) \rangle, P_0}^{(5.3.3a)}}{\boxed{\text{A.2.29}}} \frac{}{\Omega_{\text{light}} \vdash \langle 57, \text{FTA}(57) \rangle, \text{istore}[3] \cdot \text{goto}[-39] \cdot \epsilon, \{0, \dots, 55\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6b)}$$

$$(A.2.29) \quad \frac{\overline{\Omega_{\text{light}} \vdash \langle 59, \text{FTA}(59) \rangle, \text{goto}[-39], \langle P_0, S_{20} \rangle \rightarrow \langle 62, \top \rangle, P_0}^{(5.3.9a)}}{\Omega_{\text{light}} \vdash \langle 59, \text{FTA}(59) \rangle, \text{goto}[-39], \{0, \dots, 57\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle}^{(5.2.6a)}$$

A.3 Lightweight Certification Proof

This section gives the details of the proof for the lightweight bytecode certification Proposition 6.5.1 (on page 149). We assume the same definitions as in that proof and allow shorthands as in the previous section except in this section we allow \Rightarrow and \rightarrow for their 'certify'-annotated counterparts of Chapter 6. Finally we permit reference to FTA of Figure 4.5 since it provides just the right the frame type assignments in most cases.

$$(A.3.5) \quad \frac{\frac{\frac{\Omega \vdash 10, \text{pop}, \langle P_7, \emptyset \rangle \rightarrow 11, \text{FTA}(11), \langle P_{10}, \emptyset \rangle}{\text{A.3.6}}}{\Omega \vdash 11, \text{new}[2], \langle P_{10}, \emptyset \rangle \rightarrow 14, \text{FTA}(14), \langle P_{10}, \emptyset \rangle}}{\Omega \vdash \langle 10, \top \rangle, \text{pop} \cdot \text{new}[2] \cdot \text{CS}_{11}, \{0, 2, 5, 7\}, \langle P_{10}, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle} \quad (6.3.2a) \quad (6.2.9b)$$

where $P_{10} = P_7 \sqcap \{10 \mapsto \top\}$.

$$(A.3.6) \quad \frac{\frac{\frac{\frac{\langle \Omega, \text{LT}_{11}, \text{false} \rangle \vdash 11, \epsilon, \text{ET}, \langle P_{10}, \emptyset \rangle \rightarrow \langle P_{10}, \emptyset \rangle}{\Omega \vdash 11, \text{new}[2], \langle P_{10}, \emptyset \rangle \rightarrow 14, \text{FTA}(14), \langle P_{10}, \emptyset \rangle}}{\text{A.3.7}}}{\Omega \vdash 11, \text{new}[2] \cdot \text{athrow} \cdot \text{CS}_{14}, \{0, \dots, 10\}, \langle P_{10}, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle} \quad (6.4.4b) \quad (6.3.5a) \quad (6.2.9b)$$

where LT_{11} is defined by $\text{FTA}(11) = \langle -, \text{LT}_{11} \rangle$.

$$(A.3.7) \quad \frac{\frac{\frac{\frac{\frac{\frac{\text{CH} \vdash \text{Abort}, \text{EA}}{\text{CH} \vdash \text{true}, \text{Abort}, \text{EA}}}{\langle \Omega, \text{LT}_{14}, \text{true} \rangle \vdash 14, \epsilon, \text{ET}, \langle P_{10}, \emptyset \rangle \rightarrow \langle P_{10}, \emptyset \rangle}}{\langle \Omega, \text{LT}_{14}, \text{true} \rangle \vdash 14, \text{Abort}, \epsilon, \langle P_{10}, \emptyset \rangle \rightarrow \langle P_{10}, \emptyset \rangle}}{\langle \Omega, \text{LT}_{14}, \text{true} \rangle \vdash 14, \text{Abort}, \text{ET}, \langle P_{10}, \emptyset \rangle \rightarrow \langle P_{10}, \emptyset \rangle}}{\Omega \vdash 14, \text{athrow}, \langle P_{10}, \emptyset \rangle \rightarrow 15, \top, \langle P_{10}, \emptyset \rangle}}{\text{A.3.8}}}{\Omega \vdash 14, \text{athrow} \cdot \text{ldc}_w[3] \cdot \text{CS}_{15}, \{0, \dots, 11\}, \langle P_{10}, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle} \quad (4.4.20b) \quad (4.4.19b) \quad (6.4.4c) \quad (6.4.4b) \quad (6.4.4d) \quad (6.3.10b) \quad (6.2.9b)$$

where LT_{14} is defined by $\text{FTA}(14) = \langle -, \text{LT}_{14} \rangle$.

$$(A.3.8) \quad \frac{\frac{\frac{\frac{\langle \Omega, \text{LT}_{15}, \text{false} \rangle \vdash 15, \epsilon, \text{ET}, \langle P_0, \emptyset \rangle \rightarrow \langle P_0, \emptyset \rangle}{\Omega \vdash 15, \text{ldc}_w[3], \langle P_{10}, \emptyset \rangle \rightarrow 18, \text{FTA}(18), \langle P_0, \emptyset \rangle}}{\text{A.3.9}}}{\Omega \vdash 15, \text{ldc}_w[3] \cdot \text{istore}[3] \cdot \text{CS}_{18}, \{0, \dots, 14\}, \langle P_0, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle} \quad (6.4.4b) \quad (6.3.5a) \quad (6.2.9b)$$

where LT_{15} is defined by $\text{FTA}(15) = \langle -, \text{LT}_{15} \rangle$.

$$(A.3.9) \quad \frac{\frac{\frac{\Omega \vdash 18, \text{istore}[3], \langle P_0, \emptyset \rangle \rightarrow 20, \text{FTA}(20), \langle P_0, \emptyset \rangle}{\text{A.3.10}}}{\Omega \vdash 18, \text{istore}[3] \cdot \text{iload}[2] \cdot \text{CS}_{20}, \{0, \dots, 15\}, \langle P_0, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle} \quad (6.3.3a) \quad (6.2.9b)$$

$$(A.3.10) \quad \frac{\frac{\overline{\Omega \vdash 20, \text{iload}[2], \langle P_0, \{20\} \rangle \rightarrow 22, \text{FTA}(22), \langle P_0, \emptyset \rangle}}{\text{A.3.11}}}{\Omega \vdash 20, \text{iload}[2] \cdot \text{iload}[3] \cdot \text{CS}_{22, \{0, \dots, 18\}}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.3a) \quad (6.2.9b)$$

$$(A.3.11) \quad \frac{\frac{\overline{\Omega \vdash 22, \text{iload}[3], \langle P_0, \{20\} \rangle \rightarrow 24, \text{FTA}(24), \langle P_0, \emptyset \rangle}}{\text{A.3.12}}}{\Omega \vdash 22, \text{iload}[3] \cdot \text{isub} \cdot \text{CS}_{24, \{0, \dots, 20\}}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.3a) \quad (6.2.9b)$$

$$(A.3.12) \quad \frac{\frac{\overline{\Omega \vdash 24, \text{isub}, \langle P_0, \{20\} \rangle \rightarrow 25, \text{FTA}(25), \langle P_0, \emptyset \rangle}}{\text{A.3.13}}}{\Omega \vdash 24, \text{isub} \cdot \text{istore}[4] \cdot \text{CS}_{25, \{0, \dots, 22\}}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.2a) \quad (6.2.9b)$$

$$(A.3.13) \quad \frac{\frac{\overline{\Omega \vdash 25, \text{istore}[4], \langle P_0, \{20\} \rangle \rightarrow 27, \text{FTA}(27), \langle P_0, \emptyset \rangle}}{\text{A.3.14}}}{\Omega \vdash 25, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{27, \{0, \dots, 24\}}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.3a) \quad (6.2.9b)$$

$$(A.3.14) \quad \frac{\frac{\overline{\Omega \vdash 27, \text{iload}[3], \langle P_0, \{20\} \rangle \rightarrow 29, \text{FTA}(29), \langle P_0, \emptyset \rangle}}{\text{A.3.15}}}{\Omega \vdash 27, \text{iload}[3] \cdot \text{ifle}[+10] \cdot \text{CS}_{29, \{0, \dots, 25\}}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.3a) \quad (6.2.9b)$$

$$(A.3.15) \quad \frac{\frac{\overline{\Omega \vdash 29, \text{ifle}[+10], \langle P_0, \{20\} \rangle \rightarrow 32, \text{FTA}(32), \langle P_{29}, \emptyset \rangle}}{\text{A.3.16}}}{\Omega \vdash 29, \text{ifle}[+10] \cdot \text{iload}[4] \cdot \text{CS}_{32, \{0, \dots, 27\}}, \langle P_{29}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.8b) \quad (6.2.9b)$$

where $P_{29} = P_0 \sqcap \{39 \mapsto \text{FTA}(39)\}$.

$$(A.3.16) \quad \frac{\frac{\overline{\Omega \vdash 32, \text{iload}[4], \langle P_{29}, \{20\} \rangle \rightarrow 34, \text{FTA}(34), \langle P_{29}, \emptyset \rangle}}{\text{A.3.17}}}{\Omega \vdash 32, \text{iload}[4] \cdot \text{istore}[2] \cdot \text{CS}_{34, \{0, \dots, 29\}}, \langle P_{29}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle}} \quad (6.3.3a) \quad (6.2.9b)$$

$$(A.3.17) \quad \frac{\overline{\Omega \vdash 34, \text{istore}[2], \langle P_{29}, \{20\} \rangle \rightarrow 36, \text{FTA}(36), \langle P_{29}, \emptyset \rangle} \quad (6.3.3a)}{\boxed{\text{A.3.18}}} \quad (6.2.9b)$$

$$\Omega \vdash 34, \text{istore}[2] \cdot \text{goto}[-16] \cdot \text{CS}_{36}, \{0, \dots, 32\}, \langle P_{29}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

$$(A.3.18) \quad \frac{\overline{\Omega \vdash 36, \text{goto}[-16], \langle P_{29}, \{20\} \rangle \rightarrow 39, \top, \langle P_{29}, \{20\} \rangle} \quad (6.3.9a)}{\boxed{\text{A.3.19}}} \quad (6.2.9b)$$

$$\Omega \vdash 36, \text{goto}[-16] \cdot \text{iload}[4] \cdot \text{CS}_{39}, \{0, \dots, 34\}, \langle P_{29}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

$$(A.3.19) \quad \frac{\overline{\Omega \vdash 39, \text{iload}[4], \langle P_{29}, \{20\} \rangle \rightarrow 41, \text{FTA}(41), \langle P_0, \{20\} \rangle} \quad (6.3.3a)}{\boxed{\text{A.3.20}}} \quad (6.2.9b)$$

$$\Omega \vdash 39, \text{iload}[4] \cdot \text{ifne}[+6] \cdot \text{CS}_{41}, \{0, \dots, 36\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

$$(A.3.20) \quad \frac{\overline{\Omega \vdash 41, \text{ifne}[+6], \langle P_0, \{20\} \rangle \rightarrow 44, \text{FTA}(44), \langle P_{41}, \{20\} \rangle} \quad (6.3.8b)}{\boxed{\text{A.3.21}}} \quad (6.2.9b)$$

$$\Omega \vdash 41, \text{ifne}[+6] \cdot \text{iload}[2] \cdot \text{CS}_{44}, \{0, \dots, 39\}, \langle P_{41}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

where $P_{41} = P_0 \sqcap \{47 \mapsto \text{FTA}(47)\}$ here and below.

$$(A.3.21) \quad \frac{\overline{\Omega \vdash 44, \text{iload}[2], \langle P_{41}, \{20\} \rangle \rightarrow 46, \text{FTA}(46), \langle P_{41}, \{20\} \rangle} \quad (6.3.3a)}{\boxed{\text{A.3.22}}} \quad (6.2.9b)$$

$$\Omega \vdash 44, \text{iload}[2] \cdot \text{ireturn} \cdot \text{CS}_{46}, \{0, \dots, 41\}, \langle P_{41}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

$$(A.3.22) \quad \frac{\overline{\Omega \vdash 46, \text{ireturn}, \langle P_{41}, \{20\} \rangle \rightarrow \langle 47, \top \rangle, \langle P_{41}, \{20\} \rangle} \quad (6.3.11b)}{\boxed{\text{A.3.23}}} \quad (6.2.9b)$$

$$\Omega \vdash 46, \text{ireturn} \cdot \text{iload}[2] \cdot \text{CS}_{47}, \{0, \dots, 44\}, \langle P_{41}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

$$(A.3.23) \quad \frac{\overline{\Omega \vdash 47, \text{iload}[2], \langle P_{41}, \{20\} \rangle \rightarrow 49, \text{FTA}(49), \langle P_0, \{20\} \rangle} \quad (6.3.3a)}{\boxed{\text{A.3.24}}} \quad (6.2.9b)$$

$$\Omega \vdash 47, \text{iload}[2] \cdot \text{istore}[4] \cdot \text{CS}_{49}, \{0, \dots, 46\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$$

$$(A.3.24) \quad \frac{\overline{\Omega \vdash 49, \text{istore}[4], \langle P_0, \{20\} \rangle \rightarrow 51, \text{FTA}(51), \langle P_0, \{20\} \rangle}}{\boxed{\text{A.3.25}}} \quad (6.3.3a)$$

$$\Omega \vdash 49, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{51}, \{0, \dots, 47\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle \quad (6.2.9b)$$

$$(A.3.25) \quad \frac{\overline{\Omega \vdash 51, \text{iload}[3], \langle P_0, \{20\} \rangle \rightarrow 53, \text{FTA}(53), \langle P_0, \{20\} \rangle}}{\boxed{\text{A.3.26}}} \quad (6.3.3a)$$

$$\Omega \vdash 51, \text{iload}[3] \cdot \text{istore}[2] \cdot \text{CS}_{53}, \{0, \dots, 49\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle \quad (6.2.9b)$$

$$(A.3.26) \quad \frac{\overline{\Omega \vdash 53, \text{istore}[2], \langle P_0, \{20\} \rangle \rightarrow 55, \text{FTA}(55), \langle P_0, \{20\} \rangle}}{\boxed{\text{A.3.27}}} \quad (6.3.3a)$$

$$\Omega \vdash 53, \text{istore}[2] \cdot \text{iload}[4] \cdot \text{CS}_{55}, \{0, \dots, 51\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle \quad (6.2.9b)$$

$$(A.3.27) \quad \frac{\overline{\Omega \vdash 55, \text{iload}[4], \langle P_0, \{20\} \rangle \rightarrow 57, \text{FTA}(57), \langle P_0, \{20\} \rangle}}{\boxed{\text{A.3.28}}} \quad (6.3.3a)$$

$$\Omega \vdash 55, \text{iload}[4] \cdot \text{istore}[3] \cdot \text{CS}_{57}, \{0, \dots, 53\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle \quad (6.2.9b)$$

$$(A.3.28) \quad \frac{\overline{\Omega \vdash 57, \text{istore}[3], \langle P_0, \{20\} \rangle \rightarrow 59, \text{FTA}(59), \langle P_0, \{20\} \rangle}}{\boxed{\text{A.3.29}}} \quad (6.3.3a)$$

$$\Omega \vdash 57, \text{istore}[3] \cdot \text{goto}[-39] \cdot \epsilon, \{0, \dots, 55\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle \quad (6.2.9b)$$

$$(A.3.29) \quad \frac{\overline{\Omega \vdash 59, \text{goto}[-39], \langle P_0, \{20\} \rangle \rightarrow 62, \top, \langle P_0, \{20\} \rangle}}{\Omega \vdash 59, \text{goto}[-39], \{0, \dots, 57\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle} \quad (6.3.9a)$$

$$(6.2.9a)$$

A.4 Example Java Source and Bytecode

The actual files used for the `cksum()` example are shown here.

Source A.4.1 (Gcd11.java). This is the Java source of `Gcd11.java` actually used for the example (from which the code shown in Example 1.3.8 was extracted).

```

1 // ON-CARD CLASSES.
2
3 // Thrown when attempting to read uninitialized credit card.
```

```

4  class UnsetCrCard extends Exception {}
5
6  // On-card class with credit card information.
7  class CrCardRd {
8      int it; // credit card number (0 if uninitialized)
9      public int getIt() throws UnsetCrCard {
10         if (it == 0) throw new UnsetCrCard();
11         return it;
12     }
13 }
14
15 // Thrown to bail out when credit card cannot be read.
16 class Abort extends Exception {}
17
18 // Check sum interface.
19 interface CkSum {
20     public int cksum(CrCardRd ccnum) throws Abort;
21 }
22
23 // DOWNLOADED CLASSES.
24
25 // Implementation of check sum calculation.
26 class Gcd11 implements CkSum {
27
28     public int cksum(CrCardRd ccnum) throws Abort {
29         int x;
30         try {
31             x = ccnum.getIt();
32         } catch (UnsetCrCard e) {
33             throw new Abort();
34         }
35         int y = 11;
36         while (true) {
37             int z = x - y;
38             if (z > 0) { x = z; }
39             else if (z == 0) { return x; }
40             else { z = x; x = y; y = z; }
41         }
42     }
43
44 }

```

File A.4.2 (Gcd11.class bytecode). This is the complete output from `javap -c -verbose Gcd11` (from which the bytecode shown in Figure 3.7 was extracted).

Compiled from Gcd11.java

synchronized class Gcd11 extends java.lang.Object implements CkSum

```

    /* ACC_SUPER bit set */
    {
        public int cksum(CrCardRd);
    /* Stack=2, Locals=5, Args_size=2 */
        Gcd11();
    /* Stack=1, Locals=1, Args_size=1 */
    }

```

Method int cksum(CrCardRd)

```

    0 aload_1
    1 invokevirtual #9 <Method int getIt()>
    4 istore_2
    5 goto 17
    8 pop
    9 new #1 <Class Abort>
    12 dup
    13 invokespecial #7 <Method Abort()>
    16 athrow
    17 bipush 11
    19 istore_3
    20 iload_2
    21 iload_3
    22 isub
    23 istore 4
    25 iload 4
    27 ifle 36
    30 iload 4
    32 istore_2
    33 goto 20
    36 iload 4
    38 ifne 43
    41 iload_2
    42 ireturn
    43 iload_2
    44 istore 4
    46 iload_3
    47 istore_2
    48 iload 4
    50 istore_3
    51 goto 20

```

Exception table:

```

    from    to    target type

```

```
0    5    8    <Class UnsetCrCard>
```

```
Method Gcd11()
```

```
0  aload_0
1  invokespecial #8 <Method java.lang.Object()>
4  return
```

A.5 Execution of the lbv Program on Gcd11

We will execute the lbv program of Chapter 7 with

- Gcd11.class is class file shown above,
- Gcd11.ch contains class hierarchy of Figure 4.2, and
- Gcd11.cksum.cert contains the certificate of Example 5.5.1.

File A.5.1 (Output of lbv on Gcd11). Shows the contents of the class hierarchy file Gcd11.ch, the certificate file Gcd11.cksum.cert, and that the constructor method “<init>” cannot be verified (because it contains an unsupported instruction). The file was produced by running the command “java -jar lbv.jar -v Gcd11.”

LIGHTWEIGHT BYTECODE VERIFICATION of 'Gcd11':

```
CLASS HIERARCHY ch = { Ljava/lang/Object; <: bot
, Ljava/lang/Throwable; <: Ljava/lang/Object;
, Ljava/lang/Exception; <: Ljava/lang/Throwable;
, Ljava/lang/RuntimeException; <: Ljava/lang/Exception;
, Ljava/lang/NullPointerException; <: Ljava/lang/RuntimeException;
, Ljava/lang/ArrayIndexOutOfBoundsException; <: Ljava/lang/RuntimeException;
, Ljava/lang/ArrayStoreException; <: Ljava/lang/RuntimeException;
, Ljava/lang/NegativeArraySizeException; <: Ljava/lang/RuntimeException;
, Ljava/lang/ClassCastException; <: Ljava/lang/RuntimeException;
, LCrCardRd; <: Ljava/lang/Object;
, LUnsetCrCard; <: Ljava/lang/Exception;
, LAbort; <: Ljava/lang/Exception;
, LckSum; <: Ljava/lang/Object;
, LGcd11; <: LckSum;
}
```

METHOD 'cksum':

```
CERTIFICATE ce =({20},
  {}),
  2)
```

VERIFICATION fta =

```

- <*, LGcd11;.LCrCardRd;.bot.bot.bot>
0: aload 1
- <LCrCardRd;, LGcd11;.LCrCardRd;.bot.bot.bot>
1: invokevirtual methodref(LCrCardRd;, methsig(getIt, []), I)
- <I, LGcd11;.LCrCardRd;.bot.bot.bot>
4: istore 2
- <*, LGcd11;.LCrCardRd;.I.bot.bot>
5: goto +12
- <LUnsetCrCard;, LGcd11;.LCrCardRd;.bot.bot.bot>
8: pop
- <*, LGcd11;.LCrCardRd;.bot.bot.bot>
9: new LAbort;
- <LAbort;, LGcd11;.LCrCardRd;.bot.bot.bot>
12: dup
- <LAbort;.LAbort;, LGcd11;.LCrCardRd;.bot.bot.bot>
13: invokespecial methodref(LAbort;, methsig(<init>, []), V)
- <LAbort;, LGcd11;.LCrCardRd;.bot.bot.bot>
16: athrow
- <*, LGcd11;.LCrCardRd;.I.bot.bot>
17: iconst 0
- <I, LGcd11;.LCrCardRd;.I.bot.bot>
19: istore 3
- <*, LGcd11;.LCrCardRd;.I.I.bot>
20: iload 2
- <I, LGcd11;.LCrCardRd;.I.I.bot>
21: iload 3
- <I.I, LGcd11;.LCrCardRd;.I.I.bot>
22: isub
- <I, LGcd11;.LCrCardRd;.I.I.bot>
23: istore 4
- <*, LGcd11;.LCrCardRd;.I.I.I>
25: iload 4
- <I, LGcd11;.LCrCardRd;.I.I.I>
27: ifle +9
- <*, LGcd11;.LCrCardRd;.I.I.I>
30: iload 4
- <I, LGcd11;.LCrCardRd;.I.I.I>
32: istore 2
- <*, LGcd11;.LCrCardRd;.I.I.I>
33: goto -13
- <*, LGcd11;.LCrCardRd;.I.I.I>
36: iload 4
- <I, LGcd11;.LCrCardRd;.I.I.I>
38: ifne +5

```

```

- <*, LGcd11;.LCrCardRd;.I.I.I>
41: iload 2
- <I, LGcd11;.LCrCardRd;.I.I.I>
42: ireturn
- <*, LGcd11;.LCrCardRd;.I.I.I>
43: iload 2
- <I, LGcd11;.LCrCardRd;.I.I.I>
44: istore 4
- <*, LGcd11;.LCrCardRd;.I.I.I>
46: iload 3
- <I, LGcd11;.LCrCardRd;.I.I.I>
47: istore 2
- <*, LGcd11;.LCrCardRd;.I.I.I>
48: iload 4
- <I, LGcd11;.LCrCardRd;.I.I.I>
50: istore 3
- <*, LGcd11;.LCrCardRd;.I.I.I>
51: goto -31
- top
VERIFICATION OF 'cksum' OK.

```

```

METHOD '<init>':
CERTIFICATE ce =({},
  {},
  2)
VERIFICATION fta =
- <*, LGcd11;>
0: aload 0
- <LGcd11;, LGcd11;>
1: invokespecial methodref(Ljava/lang/Object;, methsig(<init>, []), V)
- <*, LGcd11;>
4: return
- top
VERIFICATION OF '<init>' OK.

```

List of Judgment Signatures

4.1.10	$\text{ClassHier} \vdash_{\text{bv}} \widehat{\text{Type}}_{\perp}^{\top} \sqsubseteq \widehat{\text{Type}}_{\perp}^{\top}$	53
4.1.28	$\text{ClassHier} \vdash_{\text{bv}} (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top} \sqsubseteq_{\text{MS,ML}} (\text{FrameType}_{\text{MS,ML}})_{\perp}^{\top}$	59
4.1.30	$\text{ClassHier} \vdash_{\text{bv}} \text{StackType}_{\text{MS}} \sqsubseteq_{\text{MS}} \text{StackType}_{\text{MS}}$	60
4.1.30	$\text{ClassHier} \vdash_{\text{bv}} \text{LocalType}_{\text{ML}} \sqsubseteq_{\text{ML}} \text{LocalType}_{\text{ML}}$	60
4.2.7	$\text{StdContext} \vdash_{\text{bv}} \text{Method}, \text{FrameTypeAssign}$	65
4.2.8	$\text{CodeContext} \vdash_{\text{bv}} \text{PPoint}, \text{Code}, \text{PPoints} \xrightarrow{\text{tsafe}} \text{PPoints}$	66
4.3.3	$\text{CodeContext} \vdash_{\text{bv}} \text{PPoint} : \text{Ins} \xrightarrow{\text{tsafe}} \text{PPoint}$	68
4.4.11	$\text{ExcContext} \vdash_{\text{bv}} \text{PPoint}, \text{Excs}$	85
4.4.19	$\text{ClassHier} \vdash_{\text{bv}} \text{Propagate}, \text{ClassIdent}, \text{ExcAtt}$	88
4.4.20	$\text{ClassHier} \vdash_{\text{bv}} \text{ClassIdent}, \text{ExcAtt}$	88
5.2.5	$\text{StdContext} \vdash_{\text{lbv}} \text{Method}, \text{Cert}$	107
5.2.6	$\text{LightContext} \vdash_{\text{lbv}} \text{CodeStat}, \text{CodeSeq}, \text{PPoints}, \text{DelayConstr} \xrightarrow{\text{tsafe}} \text{PPoints}, \text{DelayConstr}$	108
5.3.1	$\text{LightContext} \vdash_{\text{lbv}} \text{CodeStat} : \text{Ins}, \text{DelayConstr} \xrightarrow{\text{tsafe}} \text{CodeStat}, \text{Pending}$	110
5.4.4	$\text{ExcContext} \vdash_{\text{lbv}} \text{PPoint}, \text{Excs}$	117
6.2.4	$\text{StdContext} \vdash_{\text{lbc}} \text{Method}, \text{FrameTypeApprox}, \text{Cert}$	127
6.2.9	$\text{CodeContext} \vdash_{\text{lbc}} \text{PPoint}, \text{CodeSeq}, \text{PPoints}, \text{PCert} \xrightarrow{\text{certify}} \text{PPoints}, \text{PCert}$	129
6.3.1	$\text{CodeContext} \vdash_{\text{lbc}} \text{PPoint} : \text{Ins}, \text{PreConstr} \xrightarrow{\text{certify}} \text{PPoint}, \text{FrameType}, \text{PreConstr}$	133
6.4.3	$\text{ExcContext} \vdash_{\text{lbc}} \text{PPoint}, \text{Excs}, \text{ExcHandlers}, \text{PreConstr} \xrightarrow{\text{certify}} \text{PreConstr}$	144

List of Definitions, etc.

1.2.2	Definition (Standard Bytecode Verification)	13
1.2.4	Remark (Digital signatures)	14
1.2.6	Observation (Bytecode verification portability.)	16
1.3.1	Definition (JVM Type Safety)	17
1.3.3	Observation (Type safety condition)	18
1.3.4	Definition (Java Lightweight Verification Concepts)	19
1.3.7	Definition (Memory Model)	20
1.3.8	Example (Checksum Computation)	20
2.1.1	Definition (Boolean Logic)	24
2.1.2	Definition (Sets)	24
2.1.3	Definition (Integers)	24
2.2.1	Definition (Sorts Declarations)	25
2.2.3	Definition (Tuples and Sequences)	25
2.2.4	Definition (Functions)	25
2.3.1	Definition (Partial Ordering)	26
2.3.2	Definition (Partially Ordered Sets)	26
2.3.3	Definition (Join and Meet)	26
2.3.4	Definition (Lattice)	26
2.3.5	Proposition (Preservation of Lattice Property)	27
2.3.6	Proposition (Completion of semicomplete lattice)	27
2.3.7	Example (Partially Ordered Sets)	27
2.3.8	Notation (Function Meet and Join)	27
2.4.1	Definition (Judgments)	27
3.1.1	Observation (The JVM type system)	30
3.1.2	Definition (A representative data-type subset)	30
3.1.3	Definition (Omitted data-types)	30
3.1.4	Definition (The Formal Target Machine)	31
3.1.5	Notation (Instruction representation)	31
3.1.6	Notation (Informal Description Parameters)	32
3.1.7	Notation (How to read the tables)	32
3.1.8	Definition (Stack Instructions)	32
3.1.9	Definition (Local Variable Instructions)	32
3.1.10	Observation (Local variable invariance)	34

3.1.11	Definition (Array Instructions)	34
3.1.12	Remark (Formalization of newarray)	34
3.1.13	Definition (Constant Pool Instructions)	34
3.1.14	Remark (Formalization of new)	35
3.1.15	Definition (The Branch Instructions)	35
3.1.16	Definition (The Goto Instruction)	35
3.1.17	Definition (The athrow Instruction)	35
3.1.18	Definition (Return Instructions)	35
3.1.19	Remark (The Java Card and the J2ME instruction sets)	39
3.1.20	Example (The checksum bytecode representation.)	39
3.2.1	Definition (The Formal Verification Context)	41
3.2.2	Definition (Constant Pool Items)	42
3.2.4	Notation (The class hierarchy subscript)	42
3.2.5	Example (The checksum class file verification)	43
3.2.6	Definition (A Method)	43
3.2.8	Definition (Bytecode Formalization)	44
3.2.10	Definition (Bytecode Sequence Program Points)	44
3.2.12	Definition (The Exception Handler Table)	45
3.2.14	Example (The checksum method formalization)	45
3.3.1	Definition (The Class Hierarchy)	46
3.3.4	Lemma (The subtype semi-lattice property)	47
3.3.6	Example (The checksum class hierarchy formalization)	47
4.1.1	Example (Type safety for method invocation)	50
4.1.2	Example (Type safety for object assignment)	50
4.1.3	Observation (Type safety for dynamic type assignment)	51
4.1.5	Definition (General Verification Constraints)	51
4.1.6	Definition (Type assignment compatibility)	52
4.1.7	Definition (The Abstracted Type Sort)	52
4.1.10	Definition (Type Assignment Compatibility)	53
4.1.11	Definition (The Type Assignment Compatibility Relation)	55
4.1.12	Lemma (Type compatibility as a partial order)	55
4.1.13	Example (The checksum compatibility ordering)	55
4.1.14	Proposition (The extended type lattice)	55
4.1.15	Remark (Multi-dimensional arrays)	57
4.1.16	Remark (Interfaces)	57
4.1.18	Definition (Frame Types)	57
4.1.19	Notation (Stack types)	57
4.1.20	Notation (Local types)	58
4.1.21	Definition (Stack Type Size)	58
4.1.22	Definition (Local Type Size)	58
4.1.24	Definition (Bounded Frame Types)	58
4.1.25	Definition (The Frame Type Sort)	59
4.1.27	Remark (The StackType)	59

4.1.28	Definition (Frame Type Assignment Compatibility)	59
4.1.30	Definition (Stack And Local Type Assignment Compatibility)	60
4.1.32	Definition (The Frame Type Assignment Compatibility Relation)	62
4.1.33	Definition (The Stack and Local Type Assignment Compatibility Relation)	62
4.1.34	Lemma (Frame type compatibility as a partial order)	62
4.1.36	Proposition (The frame type lattice property)	62
4.1.37	Corollary (The “meet” and “join” frame type existence)	63
4.1.38	Lemma (Algebraic properties)	63
4.2.2	Definition (Frame Type Typing)	63
4.2.3	Definition (Invocation Frame Type)	64
4.2.4	Example (The Checksum Invocation Frame Type)	64
4.2.5	Definition (The Formal Verification Assumptions)	64
4.2.6	Definition (The Verification Contexts)	65
4.2.7	Definition (Method Standard Verification)	65
4.2.8	Definition (Instruction Sequence Verification)	66
4.3.1	Notation (The Verification Table Notation)	67
4.3.2	Definition (The Expected Frame Type)	67
4.3.3	Definition (The Instruction Verification Signature)	68
4.3.4	Notation (Assumptions and notational conventions)	68
4.3.6	Definition (Stack Instruction Verification)	69
4.3.7	Discussion (The Null type)	70
4.3.8	Definition (Local Variable Instruction Verification)	70
4.3.9	Definition (Launched Exception Formalization)	70
4.3.10	Notation (Exception Representation)	71
4.3.11	Definition (Array Instruction Verification)	71
4.3.12	Discussion (An array of “any type”)	72
4.3.13	Definition (Simple Access, Constant Pool Instruction Verification)	73
4.3.14	Remark (The constant pool index)	74
4.3.15	Remark (The checkcast instruction)	74
4.3.16	Definition (Field Access, Constant Pool Instruction Verification)	74
4.3.17	Definition (Method Invocation, Constant Pool Instruction Verification)	75
4.3.18	Definition (A Representative Exception Subset)	76
4.3.19	Remark (Omitted exceptions)	76
4.3.20	Definition (Branch Instruction Verification)	76
4.3.21	Remark (Jump targets)	77
4.3.22	Definition (Goto Instruction Verification)	77
4.3.24	Definition (Throw Instruction Verification)	78
4.3.25	Definition (Return Instruction Verification)	79
4.3.26	Observation (Non-fall through instruction constraints)	80
4.4.1	Discussion (Description of an exception.)	80
4.4.2	Example (Exception type confusion)	80
4.4.3	Discussion (Exception Type Safety)	81
4.4.4	Definition (Compile-time Catch Situations)	81
4.4.5	Discussion (An Exception Verification Strategy)	82

4.4.6	Discussion (Checked and Unchecked, Subset Exceptions)	83
4.4.7	Remark (Uncaught exception handling)	84
4.4.8	Definition (The Exception Verification Context)	84
4.4.9	Notation (Assumptions and notational conventions)	84
4.4.10	Proposition (Handler type-safety is well-defined)	84
4.4.11	Definition (Exception Verification Signature)	85
4.4.12	Definition (No Exception Catch)	86
4.4.14	Definition (Definite Exception Catch)	87
4.4.15	Remark (Dynamic stack check)	87
4.4.16	Definition (Potential Exception Catch)	87
4.4.17	Remark (Dynamic stack check)	88
4.4.19	Definition (Uncaught Exception Verification)	88
4.4.20	Definition (Exception Attribute Verification)	88
4.4.22	Discussion (Error verification)	89
4.5.1	Proposition (The checksum method standard verifies)	89
5.1.2	Definition (Base, Frametype Constraint Solution Set)	94
5.1.3	Proposition (The existence of a base solution set)	94
5.1.5	Example (A jump situation)	95
5.1.6	Definition (The Lightweight Assumption)	96
5.1.7	Definition (A Lightweight Certificate)	96
5.1.9	Notation (The certificate components)	97
5.1.10	Definition (The Delayed Frame Type Constraint Sort)	97
5.1.12	Definition (The Current Frame Type)	98
5.1.13	Definition (Jump Directions)	98
5.1.14	Discussion (Delayed type safety handling)	98
5.1.15	Example (A lightweight procedure)	99
5.1.18	Remark (Constraint invariants)	101
5.1.19	Discussion (A formalization strategy)	102
5.1.21	Definition (Delayed Constraint Map Modifications)	104
5.2.1	Notation (Assumptions and notational conventions)	105
5.2.2	Definition (The Code Segment State)	106
5.2.3	Definition (The Lightweight Code Context)	106
5.2.5	Definition (Method Lightweight Verification)	107
5.2.6	Definition (Instruction Sequence Lightweight Verification)	108
5.2.7	Observation (Delayed constraint invariants)	109
5.2.8	Observation (P invariant)	109
5.2.9	Remark (Tail recursion)	109
5.3.1	Definition (The Instruction Lightweight Verification Signature)	110
5.3.2	Definition (Stack, Lightweight Instruction Verification)	110
5.3.3	Definition (Local Variable, Lightweight Instruction Verification)	111
5.3.4	Definition (Array, Lightweight Instruction Verification)	111
5.3.5	Definition (Simple Access, Constant Pool Lightweight Verification)	111
5.3.6	Definition (Field Access, Constant Pool Lightweight Verification)	112

5.3.7	Definition (Method Invocation, Constant Pool Lightweight Verification)	112
5.3.8	Definition (Branch, Lightweight Instruction Verification)	113
5.3.9	Definition (The Goto Lightweight Verification)	113
5.3.10	Definition (The Throw Instruction Lightweight Verification)	114
5.3.11	Definition (The Return Instruction Lightweight Verification)	114
5.3.12	Observation (Delayed constraint invariants)	115
5.4.1	Discussion (An Exception Lightweight Verification Strategy)	115
5.4.2	Notation (Assumptions and notational conventions)	116
5.4.3	Definition (The Exception Lightweight Verification Context)	116
5.4.4	Definition (Exception Lightweight Verification Signature)	117
5.4.5	Definition (No Exception Catch)	117
5.4.6	Definition (Definite Exception Catch)	119
5.4.7	Definition (Potential Catch Situation)	119
5.4.8	Observation (lightweight verification of uncaught exceptions)	120
5.4.9	Remark (Error lightweight verification)	120
5.5.1	Proposition (cksum lightweight verifies)	120
6.1.1	Theorem (lightweight bytecode verification is safe)	121
6.1.2	Discussion (A lightweight certification strategy)	121
6.1.3	Example (A smaller frame type solution)	125
6.1.4	Definition (A Lightweight Certificate Design)	125
6.1.5	Discussion (The formalization strategy)	125
6.2.1	Notation (Assumptions and notational conventions)	126
6.2.2	Definition (The Pending Certificate Sort)	126
6.2.4	Definition (Method Certification)	127
6.2.6	Lemma (BV-LBC and LBC-LBV equivalences)	128
6.2.7	Definition (The Pre-Delayed Constraint Sort)	129
6.2.9	Definition (Instruction Sequence Certification)	129
6.2.12	Observation (Pending Constraints Invariant)	131
6.2.13	Definition (Program Point Consistency)	131
6.2.14	Lemma (Sequence Verification Equivalences)	132
6.3.1	Definition (The Instruction Certification Signature)	133
6.3.2	Definition (Stack Instruction Certification)	133
6.3.3	Definition (Local Variable Instruction Certification)	133
6.3.4	Definition (Array Instruction Certification)	133
6.3.5	Definition (Simple Access, Constant Pool Instruction Certification)	134
6.3.6	Definition (Field Access, Constant Pool Certification)	134
6.3.7	Definition (Method Invocation, Constant Pool Instruction Certification)	134
6.3.8	Definition (Branch Instruction Certification)	135
6.3.9	Definition (Goto Instruction Certification)	135
6.3.10	Definition (Throw Instruction Certification)	136
6.3.11	Definition (Return Instruction Certification)	137
6.3.12	Observation (Side condition invariant)	137
6.3.13	Observation (Type constraint invariants)	137

6.3.14	Observation (P invariant)	138
6.3.30	Observation (Side condition invariant)	141
6.3.32	Corollary (Pending invariant)	142
6.4.1	Discussion (An Exception Certification Strategy)	143
6.4.2	Notation (Assumptions and conventions)	144
6.4.3	Definition (The Exception Certification Signature)	144
6.4.4	Definition (No Exception Catch)	144
6.4.5	Definition (Definite Exception Catch)	145
6.4.6	Definition (Potential Exception Catch)	146
6.4.7	Observation (Exception attribute certification)	146
6.4.8	Remark (Error certification)	146
6.4.9	Observation (Side condition invariant)	146
6.5.1	Proposition (cksum lightweight certifies)	149
7.1.1	Observation (The lightweight verifier space bound)	150
7.2.1	Remark (Primitive types)	150
7.2.2	Source (StdContext.java)	151
7.2.3	Source (ConstPool.java)	151
7.2.4	Source (FieldRef.java)	152
7.2.5	Source (MethRef.java)	152
7.2.6	Source (MethSig.java)	152
7.2.7	Source (Method.java)	153
7.2.8	Source (ExcAtt.java)	153
7.2.9	Source (CodeAtt.java)	153
7.2.10	Source (Code.java)	154
7.2.11	Source (ExcTable.java)	154
7.2.12	Source (ClassHier.java)	154
7.3.1	Remark (Failure)	155
7.3.2	Source (Type.java)	155
7.3.3	Source (FrameType.java)	158
7.3.4	Source (FrameTypeMap.java)	163
7.3.5	Source (Pending.java)	165
7.3.6	Source (Saved.java)	165
7.3.7	Source (LightContext.java)	165
7.3.8	Source (Cert.java)	167
7.3.9	Source (MethLbv.java)	167
7.3.10	Source (MethContext.java)	168
7.3.11	Source (VerifError.java)	169
7.3.12	Source (InsLbv.java)	169
7.3.13	Remark (extensions)	175
7.4.1	Source (LBV.java)	176
7.4.2	Source (ClassInfo.java)	177
7.4.3	Source (BCEL.java)	178
7.4.4	Remark (BCEL hacks)	199

7.5.1	Documentation (Installation)	200
7.5.2	Documentation (Usage)	200
7.5.3	Example (Gcd11.ch class hierarchy file)	200
7.5.4	Example (Gcd11.cksum.cert certificate file)	201
8.1.1	Example (cksum() stack map attribute)	202
8.1.2	Definition (KVM simplifications)	202
8.1.3	Theorem (Simulation of KVM)	203
8.1.4	Remark (combining KVM with lightweight bytecode verification)	203
8.1.5	Remark (resource use of KVM's bytecode verification)	203
8.2.1	Definition (Leroy's constraints)	203
8.2.2	Example (cksum() transformed to conform to Leroy's constraints)	203
8.2.3	Theorem (Simulation of Leroy's algorithm)	203
8.2.4	Remark (resource use with Leroy's restrictions)	205
A.1.1	Notation (proof trees)	215
A.4.1	Source (Gcd11.java)	231
A.4.2	File (Gcd11.class bytecode)	232
A.5.1	File (Output of lbv on Gcd11)	234

List of Figures

1.1	Standard bytecode verification.	10
1.2	Lightweight bytecode verification.	10
1.3	The <code>cksum()</code> source program.	22
3.1	The stack instruction’s runtime behavior.	33
3.2	The local variable instruction’s runtime behavior.	33
3.3	The array instruction’s runtime behaviour.	34
3.4	The constant pool instruction’s runtime behavior.	36
3.5	The jump instruction’s runtime behavior.	37
3.6	The abrupt instruction’s runtime behavior.	38
3.7	The <code>cksum()</code> method bytecode.	40
3.8	Formalizing checksum and its verification context.	48
4.1	A general $\langle \widehat{\text{Type}}_{\perp}^{\top}, \sqsubseteq_{\text{CH}} \rangle$ ordering	54
4.2	The $\langle \widehat{\text{Type}}_{\perp}^{\top}, \sqsubseteq_{\text{CH}^{\text{ck}}} \rangle$ example ordering	56
4.3	An unsafe <code>cksum()</code> variant.	81
4.4	Type safety for exceptions raised at PP.	85
4.5	A <code>cksum()</code> frame type assignment.	90
4.6	Context for <code>cksum()</code> verification.	91
5.1	Type safety establishment at jump targets.	96
5.2	A “pre-certificate” description.	97
5.3	Altering the verification strategy.	98
5.4	A code jump situation.	99
5.5	Confluence analysis.	103
5.6	Branching analysis.	104
5.7	Confluence status.	108
5.8	Delayed type constraint updates at PP.	108
5.9	Delayed branching constraints.	110
5.10	Rule correspondences for exceptions raised at PP.	117
6.1	A “smaller” <code>cksum()</code> frame type assignment.	124
6.2	Instruction rule type safety.	138
6.3	Instruction certification updates.	139
6.4	Delayed type safety checks and updates.	142

6.5	Instruction rule correspondence.	143
6.6	Exception rule type safety.	147
6.7	Exception certification updates.	147
6.8	Exception lightweight updates and type safety.	148
6.9	No-catch exception-rule correspondence.	149
6.10	Catch exception-rule correspondance.	149
7.1	Sorts implemented by basic Java types.	151
8.1	<code>cksum()</code> that verifies with Leroy's algorithm.	204

Bibliography

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.
- [2] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs (ECOOP 2000 workshop)*, Sophia-Antipolis, France, June 2000. Technical Report 269, Fernuniversität Hagen.
- [3] D. Ancona, G. Lagorio, and E. Zucca. Java separate type checking is not safe. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs (ECOOP 2000 workshop)*, Budapest, Hungary, June 2001.
- [4] R. Anderson. Why cryptosystems fail. *Comm. of the ACM*, 37(11):32–40, November 1994.
- [5] R. Anderson. Liability and computer security: Nine principles. In *Computer Security – ESORICS 94*, number 875 in LNCS, pages 231–245. Springer-Verlag, 1996.
- [6] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Sousa, and S. Yu. Formalization in Coq of the Java card virtual machine. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs (ECOOP 2000 workshop)*, Sophia-Antipolis, France, June 2000. Technical Report 269, Fernuniversität Hagen.
- [7] E. Börger and W. Schulte. Defining the Java virtual machine as platform for provably correct java compilation. In L. Brim, J. Gruska, and J. Zlatuska, editors, *MFSC, 23rd International Symposium on Mathematical Foundations of Computer Science*, number 1450 in LNCS, pages 17–35, Brno, Czech Republic, 1998.
- [8] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries*, volume 1 of *The Java Series*. Addison Wesley Longman, second edition edition, 1999. Supplement for the Java 2 Platform, Standard Edition v1.2.
- [9] Z. Chen. *Java card technology for smart cards: architecture and programmer’s guide*. The Java Series. Addison-Wesley, 2000.
- [10] A. Coglio and A. Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and

- A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs (ECOOP 2000 workshop)*, Sophia-Antipolis, France, June 2000. Technical Report 269, Fernuniversität Hagen.
- [11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
- [12] M. Dahm. Byte code engineering. In Clemens Cap, editor, *Java-Informationen-Tage*, pages 267–277. Springer-Verlag, 1999.
- [13] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1991.
- [14] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, pages 193–205, Cambridge, Massachusetts, 1986. IEEE.
- [15] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.
- [16] S. Freund. The costs and benefits of Java bytecode subroutines. In S. Eisenbach, editor, *Formal Underpinnings of Java (an OOPSLA workshop)*, Vancouver, BC, Canada, October 1998. ACM.
- [17] S. Freund and J. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 2000.
- [18] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.
- [19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.
- [20] P. Hartel and L. Moreau. Formalizing the safety of java, the java virtual machine and java card. *ACM Computing Surveys*, 33(4):517–558, 2001.
- [21] T. Jensen, D. Le Métayer, and T. Thorn. A formalisation of visibility and dynamic loading in Java. In *ICCL '98*. IEEE, 1998. Also published as a IRISA Technical Report no 1137, October 1997.
- [22] G. Kahn. Natural semantics. Rapport 601, INRIA, Sophia-Antipolis, France, February 1987.
- [23] G. Klein and T. Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue of papers from Formal Techniques for Java Programs (ECOOP 2000 workshop).
- [24] L. Lemay and R. Cadenhead. *Teach Yourself Java2 in 21 Days*. Sams Publishing, second edition, april 2001.

- [25] X. Leroy. Java bytecode verification: an overview. In *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
- [26] X. Leroy. On-card bytecode verification for Java card. In *Proceedings e-Smart 2001*. Springer-Verlag, 2001. To appear.
- [27] X. Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.
- [28] S. Liang. Sun’s new verifier. Personal Communication (e-mail), 1999. Explains how the KVM’s verifier implements lightweight verification.
- [29] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.
- [30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.
- [31] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, 1997.
- [32] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 189–411. Oxford University Press, 1992.
- [33] R. Milner. A theory of type polymorphism in programming languages. *J. Computer and System Sciences*, 17:348–375, 1978.
- [34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, second edition, 1997.
- [35] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
- [36] P. Naughton. Java history. <http://www.ils.unc.edu/blaze/java/javahist.html>. Chronology of early Java development (1991–1995).
- [37] G. C. Necula. Proof-carrying code. In *POPL ’97—24th Annual ACM Symposium on Principles of Programming Languages*. SIGPLAN Notices, January 1997.
- [38] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI ’96—Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.
- [39] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [40] T. Nipkow and D. von Oheimb. Java_{light} is type-safe – definitely. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 161–170, San Diego, California, January 1998. ACM.

- [41] M. O’Connell. Java: The inside story. *SunWorld*, 7 1995. <http://sunsite.uakom.sk/sunworldonline/swol-07-1995/swol-07-java.html>.
- [42] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCs*, pages 119–156. Springer, 1999.
- [43] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *TOPLAS*, 22(2):340–377, 2000.
- [44] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Aarhus, Denmark, 1981.
- [45] C. Pusch. Formalizing the Java virtual machine in Isabelle/HOL. Technical Report TUM-19816, Institut für Informatik, Technische Universität München, 1998.
- [46] E. Rose. Towards bytecode verification on a Java Card. In M. Abadi, editor, *Workshop on security and languages*, Palo Alto, California, October 1997. Digital SRC.
- [47] E. Rose. Towards secure bytecode verification on a Java card. M.Sc. thesis, DIKU, University of Copenhagen, 1998.
- [48] E. Rose and K. H. Rose. Lightweight bytecode verification. In S. Eisenbach, editor, *Formal Underpinnings of Java (an OOPSLA workshop)*, Vancouver, BC, Canada, October 1998. ACM.
- [49] E. Rose and K. H. Rose. Java access protection through typing. *Concurrency and Computation: Practice and Experience*, 13(13):1125–1132, 2001. First presented a Formal Techniques for Java Programs (ECOOP 2000 workshop).
- [50] D. A. Schmidt. *Denotational Semantics, a Methodology for Language Development*. Allyn and Bacon, Boston, 1986.
- [51] R. Stärk, J. Schmid, and E. Börger. *Java and The Java Virtual Machine – Definition, Verification, Validation*. Springer-Verlag, 2001.
- [52] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998. ACM.
- [53] Sun. *Java Card 2.0 Language Subset and Virtual Machine Specification*, revision 1.0 final edition, October 1997. <ftp://ftp.javasoft.com/docs/javacard/JC20-Language.ps>.
- [54] Sun. *Java Development Kit version 1.1*, 1997. Available from <http://java.sun.com/products/jdk/1.1/>.
- [55] Sun. Java frequently asked question 1.1: Where did java come from? <http://www.ibiblio.org/javafaq/javafaq.html>, 1997.

- [56] Sun. Java 2 platform, micro edition. <http://java.sun.com/j2me>, 1999.
- [57] Sun. Java card 2.1 platform. <http://java.sun.com/products/javacard/javacard21.html>, November 1999.
- [58] Sun. J2ME connected, limited device configuration. <http://jcp.org/aboutJava/community-process/final/jsr030/index.html>, May 2000.
- [59] Sun. Java 2 platform micro edition (J2ME) technology for creating mobile devices. <http://java.sun.com/products/cldc/wp/KVMwp.pdf>, May 2000.
- [60] Sun. Secure computing with Java: Now and the future. <http://java.sun.com/marketing/collateral/security.html>, 2002. White paper.
- [61] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Models and Semantics. Elsevier Science Publishers B.V., 1994.
- [62] Verificard @ munich. <http://isabelle.in.tum.de/verificard/>, 2002.
- [63] M. Wirsing. Algebraic specification. In van Leeuwen [61], pages 675–788.

Index

- aaload, 34, 72
- aastore, 34, 72
- abrupt instructions, 79
- abstract interpretation, 17
- aconst_null, 32, 33, 69
- aload, 32, 33, 70
- anewarray, 34, 72
- antisymmetric, 26
- areturn, 35, 38
- array references, 49
- ArrayAccessIns, 34
- arraylength, 34, 72
- assignment compatible, 52
- astore, 32, 33, 70
- athrow, 35, 38, 78, 114, 136

- Backward Jump, 98
- backward labels, 97
- base, frame type constraint solution set, 94
- basic block, 89, 93
- boolean logic, 24
- $\perp \square$, 72
- \perp , 52
- $\perp \square$, 52
- branching, 104
- BranchIns, 35
- BV, 126
- bytecode-safety checker, 16

- c, 44
- CA, 44
- CatchHandle, 45
- CatchType, 45
- CE, 96
- Cert, 96
 - implementation, 167
- certificate, 12, 16, 19, 96, 126
- certificate frame types, 97
- certificate generator, 19
- certification, 96
- certifier, 19
- CH, 45, 46
- checkcast, 34, 36, 73
- checked exception, 82, 83
- checker, 16, 19
- CID, 42
- CkSum, 21
- class file verification, 64
- class files, 39
- class loader, 39
- class resolver, 39
- class type, 51
- ClassFile, 39, 41
- ClassFileContext
 - implementation, 151
- ClassHier, 41, 46
 - implementation, 154
- ClassIdent, 41, 42, 45
 - implementation, 151
- Code, 43, 44
 - implementation, 154
- code segment, 105
- code state, 105, 106
- code verification context, 65
- CodeAtt, 44
 - implementation, 153
- CodeContext, 65
- CodeSeq, 44
 - implementation, 154
- CodeStat, 106
- codomain, 25
- complete, 26
- complete lattice, 55
- concatenation, 25

- conditional code jump, 76
- confluence, 104
- conservative approach, 82
- consistent, 132
- constant pool items, 41
- ConstPool, 41
 - implementation, 151
- ConstPoolIns, 34
- contravariant, 54
- covariant, 55
- CrCardRd, 21
- CS, 44
- CST, 106
- CT, 45
- current frame type, 98, 100, 101, 122

- DC, 98
- definite catch, 82
- DelayConstr, 98
- delayed constraint set, 97
- Δ , 65
- domain, 25
- dup, 32, 33, 69
- dynamic dispatch, 50

- EA, 44
- EEPROM, 20
- EH, 45
- EHS, 45
- empty sequence, 25
- ϵ , 57
- ES, 70
- ET, 45
- exactly one, 24
- ExcAtt, 44
 - implementation, 153
- ExcContext, 84, 116
- exception type safety, 80
- exceptions, 49
- ExcHandler, 45
- ExcHandlers, 45
- Excs, 70
- ExcTable, 45
 - implementation, 154
- expected frame type, 67

- family of inference rules, 67
- field access, 73
- FieldRef, 41, 42
 - implementation, 152
- fixed point, 17
- flash memory, 20
- Forward Jump, 98
- frame type, 57
- frame type assignment compatibility, 62
- FrameType, 58
 - implementation, 158
- FrameTypeCert, 96
 - implementation, 167
- FrameTypeMap
 - implementation, 163
- FREF, 42
- FTC, 96
- FTC₀, 102, 126
- function, 25
 - join, 27
 - meet, 27
 - ordering, 26
 - partial, 25

- Γ , 41
- Gcd11, 21
- General Verification Constraints, 51
- getfield, 35, 36, 74
- goto, 35, 37, 77, 113
- GotoIns, 35
- greatest lower bound, 26

- I, 31
- iadd, 32, 33, 69
- iaload, 34, 72
- iastore, 34, 72
- iconst_0, 32, 33, 69
- iconst_1, 32, 33, 69
- ID, 42
- Identifier, 42
 - implementation, 151
- iff, 24
- ifle, 35, 37, 76
- ifne, 35, 37, 76
- ifnull, 35, 37, 76

- iload, 32, 33, 70
- incompatible, 59
- inference rule, 27
- Ins, 31
- integers, 24
- invokevirtual, 35, 36, 75
- ireturn, 35, 38
- istore, 32, 33, 70
- isub, 32, 33, 69
- Item, 41
 - implementation, 151
- join, 26
 - of function, 27
- $\sqcup_{MS,ML}$, 62
- judgment, 27
 - signature, 27
- JVM type system, 30
- Label, 96
 - implementation, 167
- Labels, 96
 - implementation, 167
- lattice, 26
 - complete, 26
- LBC, 126
- LBV, 126
- ldc_w, 34, 36, 73
- least upper bound, 26
- lifted, 26
- LightContext, 106
- Lightweight Assumption, 96
- lightweight bytecode verification, 22
- lightweight certificate, 96
- lightweight certification, 22
- lightweight code context, 106
- lightweight type certificate, 22
- lightweight verification, 9
- lightweight verifier, 19
- local type, 57
- local type assignment compatibility, 60
- local variable table type, 57
- LocalIns, 32
- LocalType, 58
 - implementation, 158
- LS, 96
- M, 43
- MaxFrame, 43, 44
 - implementation, 153
- MaxLocals, 44
 - implementation, 151
- MaxStack, 44
 - implementation, 151
- meet, 26
 - of function, 27
- $\sqcap_{MS,ML}$, 62
- memory model, 20
- MethContext, 65
 - implementation, 168
- Method, 43
 - implementation, 153
- method inheritance property, 74, 75
- method verification context, 65
- MethRef, 41, 42
 - implementation, 152
- MethSig, 42
 - implementation, 152
- MFR, 44
- ML, 44
- MREF, 42
- MS, 44
- MSIG, 42
- natural numbers, 24
- natural semantics, 59
- new, 34–36, 73
- newarray, 34
- newarray_int, 34, 72
- no catch, 82
- non-fall through, 113
- non-fall through instructions, 77
- Null, 52
- null, 49
- Object, 46
- Ω , 65
- Ω_{light} , 106
- OP, 31
- OpCode, 31

- implementation, 151
- operational, 16
- operator signature, 25
- order
 - partial, 26
 - $\sqsubseteq_{MS,ML}$, 59
 - $\sqsubseteq_{MS,ML}$, 62
 - \sqsubseteq_{ML} , 60
 - \sqsubseteq_{ML}^{CH} , 62
 - \sqsubseteq_{MS} , 60
 - \sqsubseteq_{MS}^{CH} , 62
 - \sqsubseteq_{CH} , 55
- P, 98
- P_0 , 103
- partial function, 25
- partial order, 26, 55, type compatibility 62
- PCE, 126
- PCert, 126
- PCT, 129
- Pending, 98
 - implementation, 163, 165
- pending certificate, 126
- point-wise extension, 61
- pointwise partial ordering, 26
- polymorphic, 69
- pop, 32, 33, 69
- potential catch, 82
- power set, 24
- PP, 44
- PPoint, 44
 - implementation, 151
- PPoints, 44
- PPS, 44
- PPS_C , 44
- PR, 84
- pre-delayed constraint, 129
- PreConstr, 129
- proof tree, 27
- Propagate, 84
- putfield, 34, 36, 74

- range, 25
- reflexive, 26
- relation, 26

- return, 35, 38
- ReturnIns, 35
- ReturnType, 42
 - implementation, 151
- ROM, 20
- RT, 42

- S, 98
- S_0 , 103
- Saved, 98
 - implementation, 163, 165
- scratch memory, 11, 20
- semi-lattice, 55
- set notation, 24
- signature
 - of operator, 25
 - of judgment, 27
- simple access, 73
- single point of failure, 14
- size
 - local variable table, 58
 - stack, 58
- sorts, 24
- special status, 113
- stack type, 57
- stack type assignment compatibility, 60
- StackIns, 32
- StackType, 58
 - implementation, 158
- standard bytecode verification, 22
- standard bytecode verifier, 13
- StdContext, 41
 - implementation, 151
- structural constraints, 51
- subset, 24
- sunk, 26

- T, 42
- tamper proof, 22
- τ , 53
- τ_{aref} , 53
- τ_{ob} , 53
- Θ , 84
- Θ_{light} , 116
- ThrowIns, 35

- ⊥, 52
- total function, 25
- TR, 45
- transitive, 26
- TryRange, 45
- Type, 42
 - implementation, 151, 155
- type, 53
- type assignment compatibility, 55
- type safe, 63
- type safety, 17
- typing, 63

- unchecked, 82
- unchecked exception, 83
- unconditional code jump, 77
- UnsetCrCard, Abort, 21

- verification assumptions, 64
- Verification Contexts, 65
- verification table, 67

- well-sizedness, 51
- well-typed, 63
- well-typedness, 51