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 LEROY Rapporteurs
Véronique Viguié DONZEAU-GOUGE

Vincent DANOS Examinateurs
Sophia DROSSOPOULOU
Thomas JENSEN
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 flot 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 Carying Code” (PCC), afin de pouvoir définir une technique alternative qui permet de vérifier la conformité des typages du code binaire sur les systèmes restreints en mémoire. En effet, cette technique permet d’effectuer la vérification sans appliquer de méthodes cryptographiques, 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, certificat, 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 Superieur 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 Goulbault-Larrecq at ENS Cachan (l’Ecole Normale Superieur de Cachan) in Paris, for his talented, scientific critisism 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 examinateur 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 examinateur 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 examinateur 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 Cortzen, 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 TEXnical 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
# CONTENTS

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 ............................................................ 155
   7.3 Lightweight Bytecode Verification ............................................ 156
   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 \texttt{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 \texttt{lbv} Program on \texttt{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, ressource-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 ressource-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, 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

\footnote{A standard Java compiler like Sun’s, includes bytecode verification as an integrated part.}
CHAPTER 1. INTRODUCTION

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 memory\(^2\) (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, \textit{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 \cite{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 \cite[Section 4.4]{47} 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, \textit{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 \textit{the lightweight technique provides the same type safety guarantees as a standard bytecode verifier}. An important consequence is that the technique is \textit{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 .\texttt{class} files (as long as the implementation does not use JVM features outside our subset). The program uses Markus Dahm’s BCEL \cite{12} to access .\texttt{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 \cite{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 JVM\(^3\)) 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 “sandboxing” [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”.
CHAPTER 1. INTRODUCTION

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 safe\(^4\) 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:

---

\(^4\)By 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.
1.2. MOTIVATION

- 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” 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)

\[\text{A “Curry-Howard” correspondence guarantees the existence of an isomorphism between a formal type system and a formal logic.}\]
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 form 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 definition\(^6\) [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:

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

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 point\(^7\) 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 JVM\(^8\) 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.

---

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

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

\(^8\)At 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 there 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 straightforwardly 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:

\[(1.3.6a) \text{“environment” } \vdash \text{“method”,”typeannotations”}\]

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).

---

\(^9\) Two semantics commute when it is possible to define an abstraction function $\alpha$ and a realization function $\gamma$ 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 Eraseable 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.

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

**CrCardRd:** contains and yields the credit card number.

```java
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.

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

**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 \( a \land b \) is the logical and of the two, \( a \lor b \) is the inclusive logical or of the two. \( \Rightarrow \) is the logical implication of \( b \) from \( a \), \( a \iff b \) 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:

\[
\{ e_1, e_2, \ldots \} \text{ denotes the set containing the members } e_1, e_2, \text{ etc., } e \in S \text{ is true if } e \text{ is a member of the set } S, \{ f(e) \mid p(e) \} \text{ denotes the set containing the members generated by the expression } f(e) \text{ for each } e \text{ that satisfies the predicate test } p(e), S_1 \cup S_2 \text{ denotes } \{ e \mid e \in S_1 \lor e \in S_2 \}, S_1 \cap S_2 \text{ denotes } \{ e \mid e \in S_1 \land e \in S_2 \}, S_1 \setminus S_2 \text{ denotes } \{ e \mid e \in S_1 \land e \notin S_2 \}, \text{ and } \emptyset \text{ is the empty set }\{\}. \text{ The subset test } S_1 \subseteq S_2 \text{ is true when } \forall e \in S_1 : e \in S_2, \text{ sets are equal } S_1 = S_2 \text{ when } S_1 \subseteq S_2 \land S_1 \supseteq S_2, \text{ and } S_1 \subset S_2 \text{ if } S_1 \subseteq S_2 \land \neg(S_1 \supseteq S_2). \text{ Finally, } P(S) \text{ is the “power set” of subsets } \{ S' \mid S' \subseteq S \}.\]

Definition 2.1.3 (Integers). We use the usual set of natural numbers \( n \in \mathbb{N} = \{0, 1, 2, \ldots \} \) and integers \( Z = \{\ldots, -1, 0, 1, 2, \ldots \} \) 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].
2.2. SORTS AND STRUCTURES

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

\[(2.2.1a) \quad v \in \text{Sort} \]

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:

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

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

These specify the following points:

- \(\text{FREF}, \text{FREF}'\), \(\text{FREF}^3\), 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 \(\text{CID}\), \(\text{ID}\), and \(\text{T}\) 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 \(\text{MSIG}, \text{RT}, \text{EA},\) and \(\text{CA}\), denote members of the sorts MethSig, ReturnType, ExcAtt, and CodeAtt, respectively.

**Definition 2.2.3 (Tuples and Sequences).** We use \(\langle e_1, \ldots, e_n \rangle\) to denote the \(n\)-tuple of the elements \(e_1\) through \(e_n\). For sets \(S_1, \ldots, S_n\), \(S_1 \times \cdots \times S_n\) and \(\prod_{i=1}^{n} S_i\) both denote the set of tuples \(\{ \langle e_1, \ldots, e_n \rangle \mid e_1 \in S_1 \land \cdots \land e_n \in S_n \}\). \(S^n\) 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 \(\varepsilon\) denotes the empty sequence \(\langle \rangle\) for any \(S\). Tuple concatenation is denoted \(s_1 \cdot s_2\); by abuse of notation we will write \(\varepsilon \cdot s\) and \(s \cdot \varepsilon\) for \(s\), as well as \(\varepsilon\) instead of \(\langle \varepsilon \rangle\), 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 \(\{ x_1 \mapsto f(x_1), \ldots, x_n \mapsto f(x_n) \} \) or \(\{ x_i \mapsto f(x_i) \mid i \in \{1, \ldots, n\} \}\). 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 $\langle x, y \rangle \in (\subseteq)$. Furthermore, $\subseteq$ is a partial order if it is reflexive: $\forall x \in S : x \sqsubseteq x$, antisymmetric: $\forall x, y \in S : (x \sqsubseteq y \land y \sqsubseteq x) \Rightarrow x = y$, and transitive: $\forall x, y, z \in S : (x \sqsubseteq y \land y \sqsubseteq z) \Rightarrow x \sqsubseteq z$.

**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 $S$ 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 $\sqsubseteq_S$, $\bot_S$ is the set $S$ lifted to contain the special new element $\bot_S$ and be equipped with the partial order defined by $x \sqsubseteq y$ iff $x = \bot_S \lor x \sqsubseteq y$.

2. Given $S$ is equipped with the partial order $\sqsubseteq_S$, $\top_S^-$ 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 $x \sqsubseteq S y \lor y = \top_S$.

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 $\langle x_1, x_2 \rangle \sqsubseteq \langle y_2, y_2 \rangle$ if $x_1 \sqsubseteq_1 y_1 \land x_2 \sqsubseteq_2 y_2$.

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 $\forall v \in S : (x \sqsubseteq v \land y \sqsubseteq v) \Rightarrow z \sqsubseteq v$.

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 $\forall v \in S : (v \sqsubseteq x \land v \sqsubseteq y) \Rightarrow v \sqsubseteq z$.

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 $\sqcup_{x \in S} x$ and/or $\sqcap_{x \in S} x$ 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 semilattice with a greatest element then it is a complete lattice.

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

Notation 2.3.8 (Function Meet and Join). The meet ($\sqcap$) and join ($\sqcup$) 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:

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

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

\[
\begin{array}{c}
\text{premise}_1 \cdots \text{premise}_k \\
\overline{\text{conclusion}}
\end{array}
\] (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

\[
\begin{array}{c}
\text{StdContext} \vdash_{\text{bv}} \text{Method, FrameTypeApprox}
\end{array}
\]

which formalizes that we must give inference rules for building proofs for a subset of the possible combinations of judgment terms of the form $\Gamma \vdash_{\text{bv}} M, \text{FTA}$ with $\Gamma, M,$ and FTA denoting members of the sorts StdContext, Method, and FrameTypeApprox, respectively.
The proof of Proposition 4.5.1 contains the fragment

\[(2.4.2a) \quad \frac{\text{CH}^{ck} \vdash \text{FT}^{ck} \subseteq \text{FT}_{0}^{ck}}{\Gamma^{ck} \vdash M^{ck}, \text{FTA}^{ck}} (4.2.7a)\]

\[\quad\frac{\text{A.1.2}}{\text{PPS}^{ck}} (4.2.8b)\]

where all the ck-indexed metavariables as well as the subproof \textbf{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 \texttt{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 \texttt{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 methods\textsuperscript{1} and instance fields, they do require additional syntax to be included in our formalizations. They are, however, a trivial matter to add.

\textsuperscript{1}Class methods and class initializers are syntactically simpler in that they do not contain the self-reference (\texttt{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 exists 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.
3.1. THE MACHINE SUBSET

**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.

\[
I \in \text{Ins} = \text{OpCode} \times \mathbb{Z} \cup \text{OpCode} \\
\text{Op} \in \text{OpCode} = \{ \text{dup, pop, iadd, isub, iconst}, \text{0}, \text{iconst}, \text{1, aconst}, \text{null, istore, astore, iload, aaload, aastore, iastore, iaload, aaload, anewarray}, \text{int, anewarray, arraylength, checkcast, ldc}, \text{w, new, getfield, putfield, invokevirtual, ifne, ifle, ifnull, goto, athrow, return, ireturn, areturn} \}
\]

**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 ≤ 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) \( \text{StackIns} = \{ \text{iconst}_0, \text{iconst}_1, \text{aconst}\_null, \text{dup, pop, iadd, isub} \} \)

**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) \( \text{LocalIns} = \{ \text{istore, astore, iload, aload} \} \)

We state an important property for this instruction group.
### 3.1. THE MACHINE SUBSET

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>istore 0</code></td>
<td><code>...</code></td>
<td><code>...,0</code></td>
<td>Pushes a ‘zero’ integer.</td>
</tr>
<tr>
<td><code>istore 1</code></td>
<td><code>...</code></td>
<td><code>...,1</code></td>
<td>Pushes a ‘one’ integer.</td>
</tr>
<tr>
<td><code>aconst_null</code></td>
<td><code>...</code></td>
<td><code>...,null</code></td>
<td>Pushes a ‘null’ reference.</td>
</tr>
<tr>
<td><code>dup</code></td>
<td><code>...,value</code></td>
<td><code>...,value,value</code></td>
<td>Duplicates the stack top (an integer or reference value).</td>
</tr>
<tr>
<td><code>pop</code></td>
<td><code>...,value</code></td>
<td><code>...</code></td>
<td>Removes the stack top (an integer or reference value).</td>
</tr>
<tr>
<td><code>iadd</code></td>
<td><code>...,int_2,int_1</code></td>
<td><code>...,int_1+int_2</code></td>
<td>Adds the two stack-top integers.</td>
</tr>
<tr>
<td><code>isub</code></td>
<td><code>...,int_2,int_1</code></td>
<td><code>...,int_1-int_2</code></td>
<td>Subtracts the second stack-top integer from the true stack top integer.</td>
</tr>
</tbody>
</table>

**Figure 3.1:** The stack instruction’s runtime behavior.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iload index</code></td>
<td><code>...</code></td>
<td><code>...,int</code></td>
<td>Pushes the integer from local variable number <code>index</code>.</td>
</tr>
<tr>
<td><code>astore index</code></td>
<td><code>ref,...</code></td>
<td><code>...</code></td>
<td>Pushes the top reference into local variable number <code>index</code>.</td>
</tr>
<tr>
<td><code>aload index</code></td>
<td><code>...</code></td>
<td><code>...,ref</code></td>
<td>Pushes the reference from local variable number <code>index</code>.</td>
</tr>
</tbody>
</table>

**Figure 3.2:** The local variable instruction’s runtime behavior.
### 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.

\[
\text{ArrayAccessIns} = \{ \text{iaload, aaload, iastore, iload, newarray_int, anewarray, arraylength} \}
\]

### 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 \text{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.

\[
\text{ConstPoolIns} = \{ \text{checkcast, ldc\_w, new, putfield} \}
\]
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.

\[(3.1.15a) \quad \text{BranchIns} = \{\text{ifne, ifle, ifnull}\}\]

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.

\[(3.1.16a) \quad \text{GotoIns} = \{\text{goto}\}\]

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.

\[(3.1.17a) \quad \text{ThrowIns} = \{\text{athrow}\}\]

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.

\[(3.1.18a) \quad \text{ReturnIns} = \{\text{return, ireturn, areturn}\}\]
<table>
<thead>
<tr>
<th>instruction</th>
<th>stack before</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkcast</td>
<td>... , ref</td>
<td>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 $index_1 \times 256 + index_2$.</td>
</tr>
<tr>
<td>(size)</td>
<td>⇒ ... , ref</td>
<td></td>
</tr>
<tr>
<td>ldc_w</td>
<td>...</td>
<td>Pushes the integer at the constant pool location $index_1 \times 256 + index_2$.</td>
</tr>
<tr>
<td>index</td>
<td>⇒ ... , int</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td>...</td>
<td>Pushes the class reference to the new object, given from the constant pool location $index_1 \times 256 + index_2$.</td>
</tr>
<tr>
<td>index</td>
<td>⇒ ... , ref</td>
<td></td>
</tr>
<tr>
<td>getfield</td>
<td>... , ref</td>
<td>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 $index_1 \times 256 + index_2$.</td>
</tr>
<tr>
<td>index</td>
<td>⇒ ... , value</td>
<td></td>
</tr>
<tr>
<td>putfield</td>
<td>... , ref , value</td>
<td>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 $index_1 \times 256 + index_2$.</td>
</tr>
<tr>
<td>index</td>
<td>⇒ ...</td>
<td></td>
</tr>
<tr>
<td>invokevirtual</td>
<td>... , ref , [...arg]</td>
<td>Invokes the method described at the constant pool location $index_1 \times 256 + index_2$. The stack-top arguments $arg_1, \ldots$ (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.</td>
</tr>
<tr>
<td>index</td>
<td>⇒ ... , [value]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4: The constant pool instruction’s runtime behavior.
### 3.1. THE MACHINE SUBSET

<table>
<thead>
<tr>
<th>Instruction (size)</th>
<th>Stack before ⇒ after</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>goto branch (3)</code></td>
<td>... ⇒ ...</td>
<td>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.</td>
</tr>
<tr>
<td><code>ifne branch (3)</code></td>
<td>... ⇒ ...</td>
<td>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.</td>
</tr>
<tr>
<td><code>ifle branch (3)</code></td>
<td>... ⇒ ...</td>
<td>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.</td>
</tr>
<tr>
<td><code>ifnull branch (3)</code></td>
<td>... ⇒ ...</td>
<td>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.</td>
</tr>
</tbody>
</table>

Figure 3.5: The jump instruction’s runtime behavior.
<table>
<thead>
<tr>
<th>instruction (size)</th>
<th>stack before</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return (1)</td>
<td>...</td>
<td>Indicating return from a method with return type void. Any values in the current frame are discarded.</td>
</tr>
<tr>
<td>ireturn (1)</td>
<td>...,value</td>
<td>Indicating return from a method with an integer return type value 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.</td>
</tr>
<tr>
<td>areturn (1)</td>
<td>...,ref</td>
<td>Indicating return from a method with an object reference return type ref 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.</td>
</tr>
<tr>
<td>athrow (1)</td>
<td>...,ref</td>
<td>Indicates that an exception of class instance type ref has been thrown. (A reference for Throwable 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.</td>
</tr>
</tbody>
</table>

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 subset\(^2\) 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.)

\[(3.1.20a)\]  

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.

\(^2\)Java 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 items\(^3\) 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).**

\[
\begin{align*}
(3.2.1a) \quad \Gamma \in \text{StdContext} &= \text{ClassFile} \times \text{ClassHier} \\
(3.2.1b) \quad \text{ClassFile} &= \text{ConstPool} \times \text{ClassIdent} \\
(3.2.1c) \quad \text{CP} \in \text{ConstPool} &= \mathbb{N} \overset{\text{part}}\rightarrow \text{Item} \\
(3.2.1d) \quad \text{IT} \in \text{Item} &::= \text{CID} | \text{FREF} | \text{MREF} | \text{int}
\end{align*}
\]

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.

---

\(^3\)An “item” is a field in a JVM class file structure [30, §4.1].
Definition 3.2.2 (Constant Pool Items).

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

Remark 3.2.3. Because a class name unambiguously\(^4\) 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 \( \text{`CH'} \) we indicate a specific class hierarchy as specified in Notation 3.3.3. When \( \text{`CH'} \) is used as a subscript to a sort, we indicate a restricted sort with respect to the class hierarchy indicated by \( \text{`CH'} \). The ClassIdent\(_{\text{CH}}\) sort, e.g., specifies the finite set of class names which (unambiguously) identify the classes in \( \text{`CH'} \) (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 \( \text{CH} \). 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 \text{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 \( \text{CH} \). Similar considerations apply for array types. Finally, the \text{void} method return type is specifically being formalized as the unary type element \text{void}.\(^6\)

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

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

\(^6\)JVM uses a compact string representation for \text{void}.
Example 3.2.5 (The checksum class file verification). We formalize the $\text{Gcd11}$ class file as a verification context component for the $\text{cksum()}$ method. The method was described in Example 1.3.8.

$$\text{CP} = \{1 \mapsto \text{methodref}(\text{CrCardRd, methsig}(\text{getIt, } e), \text{int}),$$

$$2 \mapsto \text{Abort},$$

$$3 \mapsto \text{int}\}$$

$$\text{CID} = \text{Gcd11}$$

From the (disassembled) $\text{cksum()}$ 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 $\text{CP}$, we let entry 1 map to the description of $\text{getIt()}$ (entry #9), and entry 2 to the description of $\text{Abort}$ (combining entry #1 for the class and #7 for the initialization). (Notice how we use $e$ to formalize that $\text{getIt()}$ has no formal parameters.) Finally, we map entry 3 to the specification of an integer constant. The self-reference tag, $\text{CID}$, is formalized by the class name $\text{Gcd11}$ 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 CodeAtt 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 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) $\text{M} \in \text{Method}_{\text{MS,ML}} = \text{MethSig} \times \text{ReturnType} \times \text{ExcAtt} \times \text{CodeAtt}_{\text{MS,ML}}$
Remark 3.2.7. The $\text{max\_stack}$ field holds the maximal number of (word-sized) elements on the operand stack, whereas the $\text{max\_locals}$ 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, $\text{ML} \geq 1$ for our subset. As we cannot extend the table index byte without the wide instruction, $\text{MaxLocals}$ is only supported up to 255.\footnote{Java Cards in fact support only 31 local variable allocations.} Because max frame constraints are assured by an initial file format check prior to bytecode verification \cite[§4.8.1]{30}, these numbers have been integrated directly.

The $\text{code}$ field of the $\text{Code\_attribute}$ structure specifies “bytecode”, and is implemented as an array of instruction bytes. It is formalized by the $\text{Code}$ 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 $\text{CodeSeq}$ sort.

Definition 3.2.8 (Bytecode Formalization).

\begin{align}
\text{(3.2.8a)} & \quad C \in \text{Code} = 1 \cdot \text{CS} \\
\text{(3.2.8b)} & \quad \text{CS} \in \text{CodeSeq} = 1^* \\
\text{(3.2.8c)} & \quad \text{PPS} \in \text{PPoints} = P(\text{PPoint}) \\
\text{(3.2.8d)} & \quad \text{PP} \in \text{PPoint} = 0..65535
\end{align}

Remark 3.2.9. The maximal code index is stored in the $\text{code\_length}$ field in four (unsigned) bytes. However, Java imposes a static constraint on the $\text{code\_length}$, which limits the number of addressable code bytes to 65535 \cite[§4.8]{30}. This limit is part of the initial file format check (prior to bytecode verification) \cite[§4.8.1]{30}. However, it is a type safety requirement that $\text{execution cannot fall off the end of the code}$ \cite[§4.9.2]{30}. 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 \cite[§4.9.2]{30} 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 $\text{code}$ array is specified as non-empty \cite[§4.8.1]{30}, we have equally integrated this lower-limit in the formalization of the $\text{Code}$ sort.

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

\begin{align}
\text{(3.2.10a)} & \quad \vdash \text{PP} : \varepsilon, \text{PPS} \rightarrow \text{PPS}
\end{align}
3.2. THE CONTEXT COMPONENTS

(3.2.10b) \[ PP' : CS, PPS' \rightarrow PPS'' \]
\[ PP : I \cdot CS, PPS \rightarrow PPS'' \]
where \( \text{len}(1) \) 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 ExcHandl sort, implements an exception handler, 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) \[ ET \in \text{ExcTable} = \text{ExcHandlers} \]
(3.2.12b) \[ EHS \in \text{ExcHandlers} = \text{ExcHandler}^* \]
(3.2.12c) \[ EH \in \text{ExcHandler} = \text{TryRange} \times \text{CatchHandle} \times \text{CatchType} \]
(3.2.12d) \[ TR \in \text{TryRange} = \text{PPoint} \times \text{PPoint} \]
(3.2.12e) \[ CH \in \text{CatchHandle} = \text{PPoint} \]
(3.2.12f) \[ CT \in \text{CatchType} = \text{ClassIdent} \]

**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 checksum() method from Example 1.3.8.

\[
M = \langle \text{MSIG, RT, EA, CA} \rangle \\
\text{MSIG} = \text{methsig}(\text{checksum, \langle CrCardRd \rangle}) \\
\text{RT} = \text{int} \\
\text{EA} = \langle \text{Abort} \rangle \\
\text{CA} = \langle \text{MFR, C, ET} \rangle
\]
where the code attribute parameter, $CA$, is listed below.

$$MFR = \langle 2, 5 \rangle$$

$$C = \text{aload}[1] \text{ invokevirtual}[1] \text{ istore}[2] \text{ goto}[+8] \text{ pop}$$
$$\text{ new}[2] \text{ athrow ldc_w[3] \text{ istore}[3] \text{ iload}[2] \text{ iload}[3]}$$
$$\text{ isub \text{ iload}[4] \text{ iload}[4] \text{ ifle}[+10] \text{ iload}[4] \text{ iload}[2]}$$
$$\text{ goto}[-16] \text{ iload}[4] \text{ ifne}[+6] \text{ iload}[2] \text{ iload}[2] \text{ ireturn}$$
$$\text{ iload}[3] \text{ goto}[-39]$$

$$ET = \langle \langle 0, 7 \rangle, 10, \text{UnsetCrCard} \rangle$$

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` 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 “$\subseteq$” 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.

\[(3.3.1a) \quad \mathcal{CH} \in \text{ClassHier} = \mathbb{P}(\text{ClassIdent})\]

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

**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 $\mathcal{CH}$ indicate an subclass ordered, finite set $\langle \mathcal{CH}, \subseteq \rangle$ 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. 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 \( \text{CH} \).

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 \( \text{CH} \). 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.

\[
\text{CH}^{ck} = \{\text{CrCardRd} \leftrightarrow \text{Object}, \text{Gcd11} \leftrightarrow \text{Object}, \text{Exception} \leftrightarrow \text{Object}, \\
\text{Abort} \leftrightarrow \text{Exception}, \text{UnsetCrCard} \leftrightarrow \text{Exception}\}
\]

where "\( \overset{1}{\rightarrow} \overset{2}{\rightarrow} \)" correspond to the parent map induced by the "\( \overset{1}{\text{extends}} \overset{2}{\text{...}} \)" Java clause, i.e., "\( \overset{1}{\leq} \overset{2}{\leq} \)". In this thesis, however, we generally adopt the somewhat counter-intuitive depiction of subclasses above superclasses as follows:

```
CrCardRd       Gcd11       Exception
    \downarrow     \downarrow     \downarrow
      Object       Abort         UnsetCrCard
```

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.
CHAPTER 3. THE FORMAL VERIFICATION CONTEXT

\[
CP = \{1 \mapsto \text{methodref}(\text{CrCardRd}, \text{methsig}(\text{getIt}, \epsilon)\text{, int)}, \\
2 \mapsto \text{Abort}, \\
3 \mapsto \text{int} \}
\]

\[
\text{CID} = \text{Gcd11}
\]

\[
\text{M} = (\text{MSIG}, \text{RT}, \text{EA}, \text{CA})
\]

\[
\text{MSIG} = \text{methsig}(\text{cksum, CrCardRd})
\]

\[
\text{RT} = \text{int}
\]

\[
\text{EA} = (\text{Abort})
\]

\[
\text{CA} = (\text{MFR, C, ET})
\]

\[
\text{MFR} = (2, 5)
\]

\[
\text{istore[3] goto[+39]}
\]

\[
\text{ET} = \langle (0, 7), 10, \text{UnsetCrCard} \rangle
\]

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

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-dimentional 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, \textit{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 principle\(^2\) as descibed by many text books [1].

\(\text{\footnotesize\(^1\)}\)“SmallTalk” is an almost pure realization of an object oriented language with no type declarations [18].

\(\text{\footnotesize\(^2\)}\)The correspondance is sometimes referred to as the \textit{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, \textit{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.

```java
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 \texttt{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 \texttt{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.

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

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 \texttt{dummy()} method is declared in class \(T\) or one of it’s superclasses (as illustrated in Example 4.1.1). Since \(x\) is a local variable, whose assigned values are stored on an \textit{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 \texttt{x.dummy()} call from failing, \textit{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 \texttt{dummy()} is declared in either \(T\) or a super class.

---

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

\(^4\)The 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 consolidates 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 $\text{CID}_{\text{stat}}$ is a statically declared class (reference) type, and $\text{CID}_{\text{dyn}}$ is the dynamically assigned (or parameter bound) class instance type. Both declared in the same class hierarchy $\text{CH}$. The following type constraint must hold:

$$\text{CID}_{\text{dyn}} \subseteq_{\text{CH}} \text{CID}_{\text{stat}}$$

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: ⊥ and ⊥[], 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 ⊥[], 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 Typeprim \(\cup\) Typeob , with Typeob given by Typecref \(\cup\) Typearef.

**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.

\[
\begin{align*}
\text{Type}_{\text{aref}} &= \{\bot\} \cup \text{Type}_{\text{aref}} \\
\text{Type}_{\text{ob}} &= \{\text{Null}\} \cup \text{Type}_{\text{cref}} \cup \text{Type}_{\text{aref}}
\end{align*}
\]

\(^5\)Milner was the first to describe analytically, the erroneous state that a variable can be in, by a special value "WRONG" [33].
(4.1.7c) \[ \text{Type}_\rightarrow = \{\bot\} \cup \{\top\} \cup \text{Type}_{\text{prim}} \cup \text{Type}_{\text{ob}} \]

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 \( \tau \) in \( \text{Type}_\rightarrow \). Specifically, we let \( \tau_{\text{ob}} \) refer to an element in \( \text{Type}_{\text{ob}} \), and \( \tau_{\text{aref}} \) to denote an element in \( \text{Type}_{\text{aref}} \).

The Notation 3.2.4 specifies that Type is defined in (3.2.2f) as a restricted set with respect to some class hierarchy \( \text{CH} \). According to Notation 3.3.3, \( \text{CH} \) 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 \( \text{Type}_\rightarrow \) 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 Definition 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:

\[
\begin{array}{c}
\text{ClassHier} \vdash_{bv} \text{Type}_\rightarrow \subseteq \text{Type}_\rightarrow
\end{array}
\]

where " \( \text{CH} \vdash \tau \subseteq \tau' \) " reads: the type \( \tau' \) is assignment compatible with the type \( \tau \) on the abstraction of the type sort Type, restricted by the class hierarchy \( \text{CH} \).

(4.1.10a) \[
\text{CH} \vdash \tau \subseteq \tau
\]

(4.1.10b) \[
\text{CH} \vdash \bot \subseteq \tau
\]

(4.1.10c) \[
\text{CH} \vdash \tau \subseteq \top
\]

(4.1.10d) \[
\text{CH} \vdash \bot \subseteq \text{int} \]

(4.1.10e) \[
\text{CH} \vdash \bot \subseteq \text{CID} \]

(4.1.10f) \[
\text{CH} \vdash \text{Object} \subseteq \bot
\]

(4.1.10g) \[
\text{CH} \vdash \text{Null} \subseteq \top
\]
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 ⊥ [] 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:

$\tau \subseteq_{\text{CH}} \tau'$

We write “$\tau \subseteq_{\text{CH}} \tau'$” with the meaning that the (uniquely) associated rule “$\text{CH} \vdash \tau \subseteq \tau'$” 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 $\subseteq_{\text{CH}}$ 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 $\langle \text{Type}_\bot, \subseteq_{\text{CH}} \rangle$ 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 $\subseteq_{\text{CH}}$, constructs a *complete lattice* on the abstracted Type, restricted by CH.

**Proof.** The ordered set $\langle (\text{Type}_\bot) \setminus \{\bot, \text{Null}\}, \subseteq_{\text{CH}} \rangle$ constructs a finite tree which is rooted in $\bot$. So, for any $\tau, \tau' \in (\text{Type}_\bot) \setminus \{\bot, \text{Null}\}$ we have that $\tau \cap \tau'$ exists. (Take the smallest subtree completion which contains $\tau$ and $\tau'$. The meet will be the root of that subtree.) Moreover we have that $\tau \cap \text{Null} = \text{Null} \cap \tau = \tau$ is defined for all $\tau \in (\text{Type}_\bot) \setminus \{\top, \text{int}\}$, and that $\text{int} \cap \text{Null} = \text{Null} \cap \text{int} = \bot$. Since $\tau \cap \tau'$ exists for all $\tau, \tau' \in (\text{Type}_\bot) \setminus \{\top\}$, we have that the set constructs a *semi-lattice* with the ordering $\subseteq_{\text{CH}}$. Finally we have that $\tau \sqsubseteq \top$ is defined for all $\tau \in (\text{Type}_\bot) \setminus \{\top\}$. That is $\top$ is a top element for the extended type sort. By lattice theoretical results, we can immediately conclude that $\langle \text{Type}_\bot, \subseteq_{\text{CH}} \rangle$ constructs a complete lattice [13, Lemma 3.20]. \(\square\)
Figure 4.2: The \( \text{TYPE} \subseteq \text{CPL} \) example ordering
4.1. ANALYSIS AND FORMALIZATION STRATEGY

The general structure of an abstracted type sort \( \langle \text{Type}_d, \subseteq \text{CH} \rangle \) is diagrammatically illustrated in Figure 4.1. We notice, that the illustration is reversed with respect to the subclass ordering \( \text{CH} \) (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 \( \text{Type}_d \), namely the “meet” type \( \tau \cap \tau' \), or the “join” type \( \tau \cup \tau' \) [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 \( \tau \) and \( \tau' \) 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 straightforwardly 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).**

\[
\begin{align*}
\text{FT} \in \text{FrameType}_{MS,ML} &= \text{StackType}_{MS} \times \text{LocalType}_{ML} \\
\text{ST} \in \text{StackType}_{MS} &= \bigcup_{n=0}^{MS} \left( \prod_{i=0}^{n} \tau_i \right), \text{where } \tau_i \in \text{Type}_d \\
\text{LT} \in \text{LocalType}_{ML} &= \prod_{i=0}^{ML} \tau_i, \text{where } \tau_i \in \text{Type}_d 
\end{align*}
\]

**Notation 4.1.19 (Stack types).** For a stack type \( \text{ST} \) of length \( k \leq MS \) which consists of elements \( \tau_i \), for \( i \in \{0,\ldots,k-1\} \), 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 \( \varepsilon \).

\(^6\) The first local variable position must contain a class (instance) type, since we only deal with instance methods.
CHAPTER 4. STANDARD VERIFICATION FORMALIZATION

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

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) $|_MS : StackType_{MS} \rightarrow \mathbb{N}$
(4.1.21b) $|\varepsilon|_MS \defeq 0$
(4.1.21c) $|ST \cdot \tau|_MS \defeq |ST|_MS + 1$

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 $ML$. Each of these is recursively defined in the stack type (and always defined, since the size of a stack is bounded).

Definition 4.1.22 (Local Type Size).

(4.1.22a) $|_ML : LocalType_{ML} \rightarrow \mathbb{N}$
(4.1.22b) $|\langle \tau_0 \rangle|_1 = 1$
(4.1.22c) $|\langle \tau_0, \ldots, \tau_{ML-2}, \tau_{ML-1} \rangle|_ML \defeq |\langle \tau_0, \ldots, \tau_{ML-2} \rangle|_{(ML-1)} + 1$, 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 $FrameType_{MS,ML}$, indexed by their maximal frame size constraints $\langle MS, ML \rangle$.

Definition 4.1.24 (Bounded Frame Types).

(4.1.24a) $FrameType_{MS,ML} = \{ \langle ST, LT \rangle \mid ST \in StackType_{MS}, LT \in LocalType_{ML} \}$
(4.1.24b) $StackType_{MS} = \{ ST \mid ST \in StackType, |ST| \leq MS \}$
(4.1.24c) $LocalType_{ML} = \{ LT \mid LT \in LocalType, |LT| = ML \}$

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.
4.1. ANALYSIS AND FORMALIZATION STRATEGY

Definition 4.1.25 (The Frame Type Sort).
(4.1.25a) \[(\text{FrameType}_{\text{MS},\text{ML}})^{\perp} = (\bot_{\text{MS},\text{ML}}) \cup (\top_{\text{MS},\text{ML}}) \cup \text{FrameType}_{\text{MS},\text{ML}}\]
where for each \((\text{MS},\text{ML})\), all frame types \(FT \in (\text{FrameType}_{\text{MS},\text{ML}})^{\perp}\) are frame type compatible with \(\bot_{\text{MS},\text{ML}}\) and \(\top_{\text{MS},\text{ML}}\) 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 \((\text{FrameType}_{\text{MS},\text{ML}})^{\perp}\), 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 \((\bot_{\text{MS}},\text{LT})\) type (e.g., when two stack type constraints are incompatible) and \(\bot_{\text{MS},\text{ML}}\) (e.g., when the complete frame type is undefined). Consequently, no undefined stack type \(\bot_{\text{MS}}\) has been added to the StackType 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 denotation [34].

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

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

(4.1.28a) \[
\text{CH} \vdash FT \subseteq_{\text{MS},\text{ML}} FT
\]

(4.1.28b) \[
\text{CH} \vdash FT_1 \subseteq_{\text{MS},\text{ML}} FT_2 \quad \text{CH} \vdash FT_2 \subseteq_{\text{MS},\text{ML}} FT_3
\]

\[
\text{CH} \vdash FT_1 \subseteq_{\text{MS},\text{ML}} FT_3
\]

(4.1.28c) \[
\text{CH} \vdash \bot \subseteq_{\text{MS},\text{ML}} FT
\]

(4.1.28d) \[
\text{CH} \vdash FT \subseteq_{\text{MS},\text{ML}} \top
\]

(4.1.28e) \[
\text{CH} \vdash ST_1 \subseteq_{\text{MS}} ST_2 \quad \text{CH} \vdash LT_1 \subseteq_{\text{ML}} LT_2
\]

\[
\text{CH} \vdash FT_1 \subseteq_{\text{MS},\text{ML}} FT_2
\]
where \( FT_1 = \langle ST_1, LT_1 \rangle \)
\( FT_2 = \langle ST_2, LT_2 \rangle \)

For our convenience we add two special cases.

\[
\begin{align*}
(4.1.28f) & \quad \text{CH} \vdash FT_1 \sqsubseteq_{\text{MS,ML}} FT_2 \quad \text{FT}_1 = \text{FT}_2 \\
(4.1.28g) & \quad \text{CH} \vdash FT_1 \sqsubseteq_{\text{MS,ML}} FT_2 \quad \text{FT}_1 \neq \text{FT}_2 
\end{align*}
\]

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:

\[
\text{ClassHier} \vdash_{bv} \text{StackType}_{\text{MS}} \sqsubseteq_{\text{MS}} \text{StackType}_{\text{MS}}
\]

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

\[
(4.1.30a) \quad \text{CH} \vdash e \sqsubseteq_{\text{MS}} e
\]

\[
(4.1.30b) \quad \text{CH} \vdash ST_1' \sqsubseteq_{\text{MS}} ST_2' \quad \text{CH} \vdash \tau_{11}(\lvert ST_1 \rvert - 1) \sqsubseteq \tau_{21}(\lvert ST_2 \rvert - 1)
\]

\[
(4.1.30b.i) \quad \text{where } \lvert ST_1 \rvert \leq \text{MS} \\
(4.1.30b.ii) \quad \lvert ST_1 \rvert = \lvert ST_2 \rvert \\
(4.1.30b.iii) \quad ST_1 = ST_1' \cdot \tau_{11}(\lvert ST_1 \rvert - 1) \\
(4.1.30b.iv) \quad ST_2 = ST_2' \cdot \tau_{21}(\lvert ST_2 \rvert - 1)
\]

The local type assignment compatibility judgement has the signature:

\[
\text{ClassHier} \vdash_{bv} \text{LocalType}_{\text{ML}} \sqsubseteq_{\text{ML}} \text{LocalType}_{\text{ML}}
\]

where “\( \text{CH} \vdash LT \sqsubseteq_{\text{ML}} LT' \)” reads: the local type \( LT' \) is local type assignment compatible with the local type \( LT \) on the ML bounded local type sort \( \text{LocalType}_{\text{ML}} \), where the individual local type constituents are elements of the same extended type sort Type, restricted by \( \text{CH} \).
4.1. ANALYSIS AND FORMALIZATION STRATEGY

(4.1.30c) \[ \text{CH} \vdash \varepsilon \sqsubseteq_0 \varepsilon \]

(4.1.30d) \[
\begin{align*}
\text{CH} \vdash \text{LT}_1' \sqsubseteq_{\text{ML}-1} \text{LT}_2' & \quad \text{CH} \vdash \tau_{12,(\text{ML}-1)} \sqsubseteq \tau_{22,(\text{ML}-1)} \\
\text{CH} \vdash \text{LT}_1 \sqsubseteq_{\text{ML}} \text{LT}_2
\end{align*}
\]

where \( |\text{LT}_1| = \text{ML} \)

(4.1.30d.i) \( |\text{LT}_1| = |\text{LT}_2| \)

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

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

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

- \( \tau_{i,1,k} \in \text{Type} \) for each \( i \in \{1,2\} \), \( k \in \{0,\ldots,(\text{ST}_i - 1)\} \), or

- \( \tau_{i,2,k} \in \text{Type} \) for each \( i \in \{1,2\} \), \( k \in \{0,\ldots,(\text{ML} - 1)\} \), respectively.

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

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 \( \text{FrameType}_{\text{MS}} \), we have decided to add these properties explicitly, since the point-wise definition does not hold on extended frame type sorts, \( (\text{FrameType}_{\text{MS,ML}}) \).

- 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 \textit{stack height} and every element on it [30, p.144]. Thus we require that \textit{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 \textit{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
```
It is tempting to suggest that \( \text{int} \in_{\text{MS}} \text{int} \cdot \text{int} \) with \( \text{int} \) 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 FrameType\(_{\text{MS},\text{ML}}\) sort, an abstraction of the Type sort, restricted by CH:

\[
\sqsubseteq_{\text{MS},\text{ML}}^{\text{CH}} \in (\text{FrameType}_{\text{MS},\text{ML}})^{\top} \times (\text{FrameType}_{\text{MS},\text{ML}})^{\perp}
\]

We write “\( \text{FT} \sqsubseteq_{\text{MS},\text{ML}}^{\text{CH}} \text{FT} \)” with the meaning that the (uniquely) associated rule “\( \text{CH} \vdash \text{FT} \sqsubseteq_{\text{MS},\text{ML}} \text{FT} \)” 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 “\( \sqsubseteq_{\text{MS},\text{ML}}^{\text{CH}} \in \text{StackType}_{\text{MS}} \times \text{StackType}_{\text{MS}} \)” 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 \( \sqsubseteq_{\text{MS},\text{ML}}^{\text{CH}} \) is a partial order relation on the frame type set \( (\text{FrameType}_{\text{MS},\text{ML}})^{\perp} \) which is abstracted from TYPE, restricted by CH.

*Proof.* Easy by observing that the frame type (carrier) set \( (\text{FrameType}_{\text{MS},\text{ML}})^{\perp} \) 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 \( (\text{FrameType}_{\text{MS},\text{ML}})^{\perp} \) is finite.

**Proposition 4.1.36 (The frame type lattice property).** The frame type assignment compatibility order \( \sqsubseteq_{\text{MS},\text{ML}}^{\text{CH}} \) constructs a complete lattice on \( (\text{FrameType}_{\text{MS},\text{ML}})^{\perp} \) 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 \( \sqcap_{\text{MS},\text{ML}} \) and \( \sqcup_{\text{MS},\text{ML}} \) are given the usual meaning [50, chapter 6].
Corollary 4.1.37 (The “meet” and “join” frame type existence). For all frame types \( FT_1, FT_2 \in (\text{FrameType}_{\text{MS,ML}}) \uparrow \), there exists a unique “meet” frame type \( FT_1 \cap_{\text{MS,ML}} FT_2 \) such that the frame type \( \{ FT_1 \cap_{\text{MS,ML}} FT_2 \} \subseteq_{\text{MS,ML}} FT_1 \), and \( \{ FT_1 \cap_{\text{MS,ML}} FT_2 \} \subseteq_{\text{MS,ML}} FT_2 \), and there exists a unique “join” frame type \( FT_1 \cup_{\text{MS,ML}} FT_2 \) such that \( \{ FT_1 \cup_{\text{MS,ML}} FT_2 \} \subseteq_{\text{MS,ML}} FT_1 \), and such that the frame type \( \{ FT_1 \cup_{\text{MS,ML}} FT_2 \} \subseteq_{\text{MS,ML}} FT_2 \).

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

Lemma 4.1.38 (Algebraic properties). We have that \( \{ \subseteq_{\text{MS,ML}}, \{ \text{FrameType}_{\text{MS,ML}} \} \uparrow \}, \cap_{\text{MS,ML}} \) and \( \{ \subseteq_{\text{MS,ML}}, \{ \text{FrameType}_{\text{MS,ML}} \} \perp \}, \cup_{\text{MS,ML}} \) are commutative and associative.

Proof. Follows immediately from the fact that \( \{ \subseteq_{\text{MS,ML}}, \{ \text{FrameType}_{\text{MS,ML}} \} \uparrow \} \) 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.

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. 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 \( \bot \).

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

\[(4.2.2a) \quad \text{FTA} \in \text{FrameTypeAssign}_{\text{MS,ML}} = \text{PPoint}_c \to \text{FrameType}_{\text{MS,ML}} \]

\(^7\)The 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 \( \varepsilon \)-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 \( MS, ML \).

\[
\begin{align*}
(4.2.3a) & \quad FT_0 \in \left( \text{FrameType}_{MS,ML} \right)_{\uparrow} \\
(4.2.3b) & \quad FT_0 = \langle \varepsilon, \langle CID_0, T_1, \ldots, T_k, \bot_{k+1}, \ldots, \bot_{ML-1} \rangle \rangle; \; 0 \leq k < ML
\end{align*}
\]

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

**Example 4.2.4 (The Checksum Invocation Frame Type).**

\[
\begin{align*}
FT_0^{ck} & \in \left( \text{FrameType}_{2,5} \right)_{\uparrow} \\
FT_0^{ck} & = \langle \varepsilon, \{ \text{Gcd11}, \text{CrCardRd}, \bot, \bot, \bot \} \rangle
\end{align*}
\]

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 \textit{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].
4.2. BYTCODE VERIFICATION

- 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 $\Delta$, called the method verification context, and $\Omega$, 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).**

\[(4.2.6a) \quad \Delta \in \text{MethContext}_{MS,ML} = \text{ReturnType} \times \text{MaxFrame}_{MS,ML} \times \text{ExcAtt} \times \text{ExcTable} \]
\[(4.2.6b) \quad \Omega \in \text{CodeContext}_{MS,ML} = \text{StdContext} \times \text{MethContext}_{MS,ML} \times \text{FrameTypeAssign}_{MS,ML} \]

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

\[
\begin{align*}
\text{StdContext} \vdash_{bv} \text{Method, FrameTypeAssign} \\
\text{CH} \vdash \text{FTA}(0) \subseteq FT_0 \\
\Omega \vdash \emptyset, C, \emptyset \overset{\text{tsafe}}{\Rightarrow} PPS \\
\Gamma \vdash_{bv} \text{M, FTA}
\end{align*}
\]

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

\[(4.2.7a) \]

(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) \( M = \langle \text{MSIG}, \text{RT}, \text{EA}, \text{CA} \rangle \)
(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) \( \text{MSIG} = \text{methsig}(\text{ID}, \text{T}_1, \ldots, \text{T}_k) \)
(4.2.7a.viii) \( 0 \leq k < \text{ML} \)
(4.2.7a.ix) \( \text{FT}_0 = \langle \epsilon, \langle \text{CID}, \text{T}_1, \ldots, \text{T}_k, \bot_{k+1}, \ldots, \bot_{\text{ML}-1} \rangle \rangle \)
(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

\[
\text{CodeContext} \vdash_{bv} \text{PPoint, Code, PPoints} \xrightarrow{\text{tsafe}} \text{PPoints}
\]

where "\( \Omega \vdash \text{PP, CS, PPS} \xrightarrow{\text{tsafe}} \text{PP's} \)" reads : the code sequence \( \text{CS} \), starting at program point \( \text{PP} \) with a set of prior, verified program points \( \text{PPS} \), is verified in the code verification context \( \Omega \), with the accumulated set of verified program points \( \text{PPS}' \).

(4.2.8a)

\[
\frac{\Omega \vdash \text{PP}, I \xrightarrow{\text{tsafe}} \text{PP}', \top}{\Omega \vdash \text{PP}, I, \text{PPS} \xrightarrow{\text{tsafe}} \text{PPS}'}
\]

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

\[
\Omega \vdash \text{PP} : I_1 \xrightarrow{\text{tsafe}} \text{PP}', \text{FT}_{\text{PP}'} \\
\text{CH} \vdash \text{FTA}(\text{PP}') \sqsubseteq \text{FT}_{\text{PP}'}
\]

(4.2.8b)

\[
\frac{\Omega \vdash \text{PP}', I_2 \cdot \text{CS}, \text{PPS} \xrightarrow{\text{tsafe}} \text{PPS}''}{\Omega \vdash \text{PP}, I_1 \cdot I_2 \cdot \text{CS}, \text{PPS} \xrightarrow{\text{tsafe}} \text{PPS}''}
\]
where \( \Omega = \langle \Gamma, \_ , \text{FTA} \rangle \)
\( \Gamma = \langle \_ , \text{CH} \rangle \)
\( \text{PPS}' = \text{PPS} \cup \{ \text{PP} \} \)

**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 PP \( \in \) 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.

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

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 \( \text{PP}' \), 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 \( \text{FT}_0 \),
- the expected frame type after a non-fall through instruction to be \( \top \).
An expected frame type is denoted by $F_T^{PP'}$ 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

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

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

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 assure 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 $\langle CP, CID \rangle$ in the verification context $\Gamma$ 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 $FTA(PP')$, and the expected frame type $F_T^{PP'}$ are well-defined. Specifically, we often use $PP''$ to signify a jump target.
- Furthermore, we have omitted the subscripts 'MS' and 'ML' from the $\subseteq$ symbol, in order to ease the readability to the extent where they are not obvious from the context.
- We use the notation $F_T^{PP'}$, 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 $FTA(PP)$ 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.
4.3. INSTRUCTION VERIFICATION

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' ∈ 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 straightforwardly 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. 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 $\text{Null}$, 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.

\[
(\text{4.3.6a})
\]

\[
\begin{array}{c}
\Omega \vdash PP : I \xrightarrow{\text{safe}} PP', FT_{PP}' \\
\end{array}
\]

where \(\Omega = \langle \Gamma, \Delta, FTA \rangle\)

(4.3.6a.i) \(FTA(PP) = \langle ST, LT \rangle\)

(4.3.6a.ii) \(FT_{PP}' = \langle ST', LT \rangle\)

(4.3.6a.iii) \(\Gamma = \langle \_, CH \rangle\)

(4.3.6a.iv) \(\Delta = \langle \_, \langle MS, \_ \rangle, \_, \_ \rangle\)

(4.3.6a.v) \(PP' = PP + 1\)

(4.3.6a.vi) \(|ST'| < MS\)

(4.3.6a.vii)

<table>
<thead>
<tr>
<th></th>
<th>ST</th>
<th>ST'</th>
</tr>
</thead>
<tbody>
<tr>
<td>$iconst_0$</td>
<td>ST</td>
<td>ST·int</td>
</tr>
<tr>
<td>$iconst_1$</td>
<td>ST</td>
<td>ST·int</td>
</tr>
<tr>
<td>$aconst_null$</td>
<td>ST</td>
<td>ST·Null</td>
</tr>
<tr>
<td>$dup$</td>
<td>$ST_1\cdot\tau$</td>
<td>$ST_1\cdot\tau\cdot\tau$</td>
</tr>
<tr>
<td>$pop$</td>
<td>$ST_1\cdot\tau$</td>
<td>$ST_1$</td>
</tr>
<tr>
<td>$iadd$</td>
<td>$ST_1\cdot\text{int}\cdot\text{int}$</td>
<td>$ST_1\cdot\text{int}$</td>
</tr>
<tr>
<td>$isub$</td>
<td>$ST_1\cdot\text{int}\cdot\text{int}$</td>
<td>$ST_1\cdot\text{int}$</td>
</tr>
</tbody>
</table>

\[\text{8A “polymorphic type” is used to denote a type variable which can be assigned a non-polymorphic (ground) type. For example used to specify an input-output function for a program.}\]
Discussion 4.3.7 (The null type). As alluded 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 by 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 straightforwardly 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.

\[(4.3.8a)\]
\[
\Omega \vdash PP : I \overset{\text{safe}}{\rightarrow} PP', FT_{PP}'
\]
\[(4.3.8a.i)\]
\[
\text{where } \Omega = \langle \Gamma, \Delta, FTA \rangle
\]
\[(4.3.8a.ii)\]
\[
FT_{PP}' = \langle ST', LT' \rangle
\]
\[(4.3.8a.iii)\]
\[
\Gamma = \langle \_ , CH \rangle
\]
\[(4.3.8a.iv)\]
\[
\Delta = \langle \_ , \langle MS, ML \rangle , \_ , \_ \rangle
\]
\[(4.3.8a.v)\]
\[
PP' = PP + 2
\]
\[(4.3.8a.vii)\]
\[
|ST'| < MS
\]
\[(4.3.8a.viii)\]
\[
n < ML
\]

(4.3.8a.ix)

<table>
<thead>
<tr>
<th>\text{Instruction}</th>
<th>\text{ST}</th>
<th>\text{ST'}</th>
<th>\text{LT'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>istore[n]</td>
<td>\text{ST'} \cdot \text{int}</td>
<td>\text{ST'}</td>
<td>\text{LT}[n \mapsto \text{int}]</td>
</tr>
<tr>
<td>astore[n]</td>
<td>\text{ST'} \cdot \tau_{ob}</td>
<td>\text{ST'}</td>
<td>\text{LT}[n \mapsto \tau_{ob}]</td>
</tr>
<tr>
<td>iload[n]</td>
<td>\text{ST}</td>
<td>\text{ST} \cdot \text{int where LT(n) = int}</td>
<td>\text{LT}</td>
</tr>
<tr>
<td>aload[n]</td>
<td>\text{ST}</td>
<td>\text{ST} \cdot \tau_{ob where LT(n) = \tau_{ob}}</td>
<td>\text{LT}</td>
</tr>
</tbody>
</table>

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.

\[(4.3.9a)\]
\[
ES \in \text{Excs} = \text{ClassIdent}^*
\]
4.3. INSTRUCTION VERIFICATION

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

\[ E_1 \cdot E_2 \ldots E_n \]

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 \( \tau \), which by convention is an element in the \( \text{Type} \) 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 \( \text{FTA} \) 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.

\[
\begin{align*}
\Theta & \vdash PP, ES, ET \\
\Omega & \vdash PP : I_{\text{tsafe}} : PP', FT_{pp}'
\end{align*}
\]

\[
\begin{align*}
(4.3.11a) & \quad \text{where} \quad \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle \\
(4.3.11a.i) & \quad \Theta = \langle \Omega, \text{LT}, \text{false} \rangle \\
(4.3.11a.ii) & \quad \text{FTA}(PP) = \langle \text{ST}, \text{LT} \rangle \\
(4.3.11a.iii) & \quad FT_{pp}' = \langle \text{ST}', \text{LT} \rangle \\
(4.3.11a.iv) & \quad \Delta = \langle \_ , \langle \text{MS}, \_ \rangle , \_ , \text{ET} \rangle \\
(4.3.11a.v) & \quad \mid \text{ST}' \mid < \text{MS}
\end{align*}
\]
Discussion 4.3.12 (An array of “any type”). We notice that the \texttt{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 \textit{observable} for our static type verification, by the addition of the special abstract type \texttt{‘\_\_\_’}, 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:

```java
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 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.

$\Theta \vdash PP, ES, ET$

$\Omega \vdash PP : I \xrightarrow{\text{safe}} PP', FT_{PP'}$

where $\Omega = \langle \Gamma, \Delta, FTA \rangle$

$\Theta = \langle \Omega, LT, \text{false} \rangle$

$FTA(PP) = \langle ST, LT \rangle$

$FT_{PP'} = \langle ST', LT \rangle$

$\Gamma = \langle \langle CP, \_ \rangle, \text{CH} \rangle$

$\Delta = \langle \_ , \langle MS, \_ \rangle , \_ , \_ , \_ , ET \rangle$

$PP' = PP + 3$

$|ST'| < MS$

<table>
<thead>
<tr>
<th>$I$</th>
<th>$ST$</th>
<th>$ST'$</th>
<th>$CP[n]$</th>
<th>$ES$</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkcast[n]</td>
<td>$ST \cdot \tau_{ob}$</td>
<td>$ST' \cdot \tau_{ob}$</td>
<td>$\tau_{ob}'$</td>
<td>ClassCastException</td>
</tr>
<tr>
<td>ldc_w[n]</td>
<td>ST</td>
<td>ST\cdot\text{int}</td>
<td>int</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>new[n]</td>
<td>ST</td>
<td>ST\cdot\text{CID}</td>
<td>CID</td>
<td>$\epsilon$</td>
</tr>
</tbody>
</table>
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 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., COD 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.

\[
\begin{align*}
\text{CH} & \vdash T \subseteq \tau \\
\Theta & \vdash PP, ES, ET \\
\Omega & \vdash PP : I \xrightarrow{tsafe} PP', FT_{pp'}
\end{align*}
\]

(4.3.16a)

where \( \Omega = \langle \Gamma, \Delta, FTA \rangle \)

(4.3.16a.i)

\( \Theta = \langle \Omega, LT, \text{false} \rangle \)

(4.3.16a.ii)

\( FTA(PP) = \langle ST, LT \rangle \)

(4.3.16a.iii)

\( FT_{PP} = \langle ST', LT \rangle \)

(4.3.16a.iv)

\( \Gamma = \langle \langle CP, _ \rangle, \text{CH} \rangle \)

(4.3.16a.v)

\( \Delta = \langle _, _ , _ , ET \rangle \)

(4.3.16a.vi)

\( PP' = PP + 3 \)

(4.3.16a.vii)

\( CID \leq_{CH} COD' \)

(4.3.16a.viii)

\( \forall \)

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.

\[
\begin{array}{|c|c|c|c|c|}
\hline
I & ST & ST' & CP[n] & ES \\
\hline
\text{putfield[n]} & ST_1 \cdot CID \cdot \tau & ST_1 & \text{fieldref(CID', \_ , T)} & \text{NullPointerException} \\
\hline
\text{getfield[n]} & ST_1 \cdot CID & ST_1 \cdot T & \text{fieldref(CID', \_ , T)} & \text{NullPointerException} \\
\hline
\end{array}
\]

\( j \)

An 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 \( \tau \) by convention is an element in 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:

\[
\begin{align*}
\text{CH} & \vdash \tau_1 \subseteq \tau_1 \\
& \vdots \\
\text{CH} & \vdash \tau_j \subseteq \tau_j \\
\Theta & \vdash \text{PP}, \text{ES}, \text{ET} \\
\Omega & \vdash \text{PP} : 1 \xrightarrow{\text{safe}} \text{PP}', \text{FT}_{\text{PP}}',
\end{align*}
\]

(4.3.17a)

where \( \Omega = \langle \Gamma, \Delta, \text{FTA} \rangle \)

(4.3.17a.i)

\( \Theta = \langle \Omega', \text{LT}, \text{false} \rangle \)

(4.3.17a.ii)

\( \text{FTA}(\text{PP}) = \langle \text{ST}, \text{LT} \rangle \)

(4.3.17a.iii)

\( \text{FT}_{\text{PP}} = \langle \text{ST}', \text{LT} \rangle \)

(4.3.17a.iv)

\( \Gamma = \langle \langle \text{CP}, \_ \rangle, \text{CH} \rangle \)

(4.3.17a.v)

\( \Delta = \langle \_ , \_ , \_ , \text{ET} \rangle \)

(4.3.17a.vi)

\( \text{PP}' = \text{PP} + 3 \)

(4.3.17a.vii)

\( \text{ST}_{\text{PP}} = \text{ST} + 3 \)

(4.3.17a.viii)

\( \text{ST}_{\text{CID} \cdot \tau_1 \ldots \tau_j} \cdot \text{methref}(\text{CID}' , \text{methsig}(\_ , \tau_1 \ldots , \tau_j , \text{void})) \)

\( \text{ST}_{\text{CID} \cdot \tau_1 \ldots \tau_j} \cdot \text{methodref}(\text{CID}' , \text{methsig}(\_ , \tau_1 \ldots , \tau_j , \text{void})) \)

(4.3.17a.ix)

\( \text{invokevirtual}[n] \cdot \text{CID} \leq \text{CH} \cdot \text{CID}' \cdot \{ E | E \leq \text{CH} \cdot \text{Throwable} \} \)

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).

\[
\text{ES_{RunExc}} \overset{\text{def}}{=} \{\text{NegativeArraySizeException, ArrayStoreException, ArrayIndexOutOfBoundsException, ClassCastException, NullPointerException}\}
\]

- 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. Statistically, 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) \[
\frac{\text{CH} \vdash \text{FTA}(PP'') \sqsubseteq F_{pp'}\text{pp'}}{\Omega \vdash PP : I \xrightarrow{\text{safe}} PP', F_{pp'}\text{pp'}}
\]

where \( \Omega = \langle \Gamma, \omega, \text{FTA} \rangle \)

(4.3.20a.i) \[
\text{FTA}(PP) = \langle ST, LT \rangle
\]

(4.3.20a.iii) \[
F_{pp'} = \langle ST', LT \rangle
\]

(4.3.20a.iv) \[
\Gamma = \langle \omega, \text{CH} \rangle
\]

(4.3.20a.v) \[
PP' = PP + 3
\]

(4.3.20a.vi) \[
PP'' = PP + n
\]

(4.3.20a.vii) \[
\begin{array}{|c|c|}
\hline
\text{I} & \text{ST} \\
\hline
\text{ifne}[n] & ST'.\text{int} \\
\text{ifle}[n] & ST'.\text{int} \\
\text{ifnull}[n] & ST'.\tau_{\text{ob}} \\
\hline
\end{array}
\]
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.

\[(4.3.22a)\quad CH \vdash FTA(PP'') \sqsubseteq FT_{PP''}^{PP} \]

\[
\Omega \vdash PP: \mathit{goto} \mapsto \mathit{safe} \rightarrow PP', T
\]

\[(4.3.22a.i)\quad \text{where} \quad \Omega = \langle \Gamma, \_\_\_, FTA \rangle\]
\[(4.3.22a.ii)\quad FT_{PP''}^{PP} = FTA(PP)\]
\[(4.3.22a.iii)\quad \Gamma = \langle \_\_\_, CH \rangle\]
\[(4.3.22a.iv)\quad PP' = PP + 3\]
\[(4.3.22a.v)\quad PP'' = PP + n\]

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 $T$ or $\bot$.

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 RunTime exception. 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 hereof), 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 Def-
nition 4.3.3.

\[
(4.3.24a) \quad \Theta \vdash PP, CID_E, ET \\
\Omega \vdash PP : athrow \xrightarrow{\text{safe}} PP', \top
\]

\((4.3.24a.i)\) \quad \Omega = \langle \Gamma, \Delta, FTA \rangle \\
\(4.3.24a.ii)\) \quad \Theta = \langle \Omega, LT, 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 \nleq_{CH} \text{Exception} \\
\(4.3.24a.viii)\) \quad PP' = PP + 1

\[
(4.3.24b) \quad \Theta \vdash PP, CID_E, ET \\
\Omega \vdash PP : athrow \xrightarrow{\text{safe}} PP', \top
\]

\((4.3.24b.i)\) \quad \Omega = \langle \Gamma, \Delta, FTA \rangle \\
\(4.3.24b.ii)\) \quad \Theta = \langle \Omega, LT, 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 \nleq_{CH} \text{RunTimeException} \\
\(4.3.24b.viii)\) \quad PP' = PP + 1

\[
(4.3.24c) \quad \Theta \vdash PP, CID_E, ET \\
\Omega \vdash PP : athrow \xrightarrow{\text{safe}} PP', \top
\]
4.3. INSTRUCTION VERIFICATION

where \( \Omega = (\Gamma, \Delta, \text{FTA}) \)

\[ (\text{4.3.24c.i}) \]

\( \Theta = (\Omega, \text{LT}, \text{false}) \)

\[ (\text{4.3.24c.ii}) \]

\( \text{FTA}(\text{PP}) = (\text{CID}_{\text{E}} \cdot \_ \cdot \_ \cdot \text{LT}) \)

\[ (\text{4.3.24c.iii}) \]

\( \Gamma = (\_ \cdot \text{CH}) \)

\[ (\text{4.3.24c.iv}) \]

\( \Delta = (\_ \cdot \_ \cdot \text{EA}, \text{ET}) \)

\[ (\text{4.3.24c.v}) \]

\( \text{CID}_{\text{E}} \subseteq_{\text{CH}} \text{RunTimeException} \)

\[ (\text{4.3.24c.vi}) \]

\( \text{PP}' = \text{PP} + 1 \)

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.

\[ (\text{4.3.25a}) \]

\[ \dfrac{\Omega \vdash \text{PP} : \text{return} \overset{\text{tsafe}}{\rightarrow} \text{PP}', \top \quad \Omega = (\_ \cdot \Delta, \text{FTA})}{(\text{4.3.25a.i})} \]

\[ \dfrac{\Delta = (\text{void}, \_ \cdot \_ \cdot \_)}{(\text{4.3.25a.ii})} \]

\[ \text{PP}' = \text{PP} + 1 \]

\[ (\text{4.3.25b}) \]

\[ \dfrac{\Omega \vdash \text{PP} : \text{ireturn} \overset{\text{tsafe}}{\rightarrow} \text{PP}', \top \quad \Omega = (\_ \cdot \Delta, \text{FTA})}{(\text{4.3.25b.i})} \]

\[ \text{FTA}(\text{PP}) = (\text{ST} \cdot \text{int}, \_ \cdot \_ \cdot \_ \cdot \_ \cdot \_) \]

\[ (\text{4.3.25b.ii}) \]

\( \Delta = (\text{int}, \_ \cdot \_ \cdot \_ \cdot \_) \)

\[ (\text{4.3.25b.iii}) \]

\( \text{PP}' = \text{PP} + 1 \)

\[ (\text{4.3.25c}) \]

\[ \dfrac{\text{CH} \vdash \text{T}_{\text{ob}} \subseteq \text{T}_{\text{ob}} \quad \text{CH} \vdash \text{PP} : \text{areturn} \overset{\text{tsafe}}{\rightarrow} \text{PP}', \top}{\Omega \vdash \text{PP} : \text{areturn} \overset{\text{tsafe}}{\rightarrow} \text{PP}', \top} \]
\[(4.3.25c.i)\] where \(\Omega = (\Gamma, \Delta, \text{FTA})\)

\[(4.3.25c.ii)\] \(\text{FTA}(\text{PP}) = (\text{ST}, \text{TOB}, -)\)

\[(4.3.25c.iii)\] \(\Gamma = (\ast, \text{CH})\)

\[(4.3.25c.iv)\] \(\Delta = (\text{TOB}, \ast, \ast, \ast)\)

\[(4.3.25c.v)\] \(\text{PP}' = \text{PP} + 1\)

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 extend 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 \(\text{FTA}(2) \nsubseteq \text{FTA}(10)\). (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: `Abort \nsubseteq \nsubseteq \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.
4.4. EXCEPTION VERIFICATION

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 declared\(^\text{10}\) 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 \(\text{CID}_E\). Furthermore, assume that an enclosing method exception handler, which cover the location where the exception can be raised, is formalized by \(\langle PP_1, PP_2, PP_0, \text{CID} \rangle\).

\(^{10}\)A declared exception is either listed by the exception attribute, or a subclass to a listed class.

<table>
<thead>
<tr>
<th>PP</th>
<th>FTA(_{PP.ST})</th>
<th>FTA(_{PP.LT}) ((\text{this} \cdot \text{ccnum} \cdot x \cdot y \cdot z))</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
<td>\text{Gcd}11 \cdot \text{CrCardRd} \cdot \bot \cdot \bot \cdot \bot</td>
<td>\text{aload}[1]</td>
</tr>
<tr>
<td>2</td>
<td>\text{CrCardRd}</td>
<td>\text{Gcd}11 \cdot \text{CrCardRd} \cdot \bot \cdot \bot \cdot \bot</td>
<td>\text{invokevirtual}[1]</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>\text{Gcd}11 \cdot \text{CrCardRd} \cdot \bot \cdot \bot \cdot \bot</td>
<td>\text{istore}[2]</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>\text{Gcd}11 \cdot \text{CrCardRd} \cdot \text{int} \cdot \bot \cdot \bot</td>
<td>\text{goto} [+8]</td>
</tr>
<tr>
<td>10</td>
<td>\text{UnsetCrCard}</td>
<td>\text{Gcd}11 \cdot \text{Abort} \cdot \bot \cdot \bot \cdot \bot</td>
<td>\text{pop}</td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>\text{Gcd}11 \cdot \text{Abort} \cdot \bot \cdot \bot \cdot \bot</td>
<td>\text{aload}[1]</td>
</tr>
<tr>
<td>13</td>
<td>\text{Abort}</td>
<td>\text{Gcd}11 \cdot \text{Abort} \cdot \bot \cdot \bot \cdot \bot</td>
<td>\text{athrow}</td>
</tr>
</tbody>
</table>

Figure 4.3: An unsafe \texttt{cksum()} variant.
1. Only when $\text{CID}_E \leq \text{CID}$, we can be sure that the handle will catch the exception at runtime. We call the situation for a definite catch.

2. Assume that $\text{CID} \leq \text{CID}_E$, i.e., $\text{CID}_E$ is a super class type of $\text{CID}$ 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 $\text{CID}_E \not\leq \text{CID}$ and $\text{CID} \not\leq \text{CID}_E$ (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 \( \text{ES}_{\text{RunExc}} \), as well as any user-declared and user-thrown subclass instance of \( \text{Exception} \). 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 \( \text{Exception} \), or any of its subclasses, which are *user-declared and user-raised*.\(^{11}\)

**Unchecked Exceptions.** Furthermore, we consider instances with compile-time type \( \text{RunTime-Exception} \) which the runtime-system may raise for our instruction subset.

\[
\text{ES}_{\text{RunExc}} \overset{\text{def}}{=} \{ \text{NegativeArraySizeException}, \\
\text{ArrayStoreException}, \\
\text{ArrayIndexOutOfBoundException}, \\
\text{ClassCastException}, \\
\text{NullPointerException} \}
\]

We notice, that this group also includes the *user-declared and user-raised* exceptions which are statically given as subclasses of \( \text{RunTimeException} \).

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

\(^{11}\)A user-raised exception is thrown by the \texttt{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-raised\textsuperscript{12} subclasses of the class Exception, e.g., EOFExeption, 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).

\begin{align*}
(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} | \text{false}
\end{align*}

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 athrow, and as we only know whether it has been user-raised when we evaluate the athrow 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 \text{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.

\textsuperscript{12}A system-raised exception is thrown by the runtime-system or the virtual machine.
4.4. EXCEPTION VERIFICATION

<table>
<thead>
<tr>
<th>Rule</th>
<th>Catch</th>
<th>Safety constraint at PP''</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4.4.12a)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12b)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12c)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12d)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12e)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.14a)</td>
<td>definite</td>
<td>CH ⊢ FTA(PP'') ⊆ FT_{PP''}</td>
</tr>
<tr>
<td>(4.4.16a)</td>
<td>potential</td>
<td>CH ⊢ FTA(PP'') ⊆ FT_{PP''}</td>
</tr>
</tbody>
</table>

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.

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.

\[ \text{ExcContext} \vdash_{by \hspace{1em} PPoint, \hspace{1em} Excs, \hspace{1em} Echs} \]

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 \( \Theta \).

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.

\[(4.4.12a)\]

\[\frac{CH \vdash_{bv} PR, CID_E, EA}{\Theta \vdash PP, CID_E \cdot ES, EHS}\]

where \(\Theta = \langle \langle \cdot, CH \rangle, \langle \cdot, EA, \epsilon \rangle, \cdot \rangle, \cdot, PR \rangle\)

\[(4.4.12b)\]

\[\frac{CH \vdash PR, CID_E, EA}{\Theta \vdash PP, ES, ET}\]

\[\frac{\Theta \vdash PP, ES, ET}{\Theta \vdash PP, CID_E \cdot ES, \epsilon}\]

where \(\Theta = \langle \langle \cdot, CH \rangle, \langle \cdot, EA, ET \rangle, \cdot \rangle, \cdot, PR \rangle\)

\[(4.4.12c)\]

\[\Theta \vdash PP, \epsilon, EHS\]

\[(4.4.12d)\]

\[\Theta \vdash PP, CID_E \cdot ES, EHS\]

\[\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS\]

where \(EH = \langle \langle PP_1, PP_2 \rangle, \cdot, \cdot \rangle\)

\[(4.4.12d.ii)\]

\[PP < PP_1 \lor PP \geq PP_2\]

\[(4.4.12e)\]

\[\Theta \vdash PP, CID_E \cdot ES, EHS\]

\[\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS\]

where \(\Theta = \langle \langle \cdot, CH \rangle, \cdot, \cdot \rangle\)

\[(4.4.12e.i)\]

\[(4.4.12e.ii)\]

\[EH = \langle \langle PP_1, PP_2 \rangle, \cdot, \cdot \rangle\]

\[PP_1 \leq PP < PP_2\]

\[(4.4.12e.iii)\]

\[CID_E \not\leq_{CH} CID_E'\]

\[(4.4.12e.iv)\]

\[CID_E' \not\leq_{CH} CID_E\]

**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 \text{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.

\[
\begin{align*}
\text{CH} & \vdash \text{FTA}(\text{PP}''') \sqsubseteq \text{FT}_{\text{pp}''} \\
\Theta & \vdash \text{PP}, \text{ES}, \text{EHS} \\
\Theta & \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}
\end{align*}
\]

(4.4.14a)

where \( \Theta = \langle \Omega, \text{LT}, \_ \rangle \)

(4.4.14a.i)

\( \Omega = \langle \langle \_ , \text{CH} \rangle, \langle \_ , \_ , \text{ET} \rangle, \text{FTA} \rangle \)

(4.4.14a.ii)

\( \text{EH} = \langle \langle \text{PP}_1, \text{PP}_2 \rangle, \text{PP}''', \text{CID}_E' \rangle \)

(4.4.14a.iii)

\( \text{FT}^\text{PP}'''' = \langle \text{CID}_E, \text{LT} \rangle \)

(4.4.14a.iv)

PP_1 \leq \text{PP} < \text{PP}_2

(4.4.14a.v)

\( \text{CID}_E \leq_{\text{CH}} \text{CID}_E' \)

(4.4.14a.vi)

**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.

\[
\begin{align*}
\text{CH} & \vdash \text{FTA}(\text{PP}''') \sqsubseteq \text{FT}_{\text{pp}''} \\
\Theta & \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS} \\
\Theta & \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}
\end{align*}
\]

(4.4.16a)

where \( \Theta = \langle \Omega, \text{LT}, \_ \rangle \)

(4.4.16a.i)

\( \Omega = \langle \langle \_ , \text{CH} \rangle, \langle \_ , \_ , \text{FT} \rangle \rangle \)

(4.4.16a.ii)

\( \text{EH} = \langle \langle \text{PP}_1, \text{PP}_2 \rangle, \text{PP}''', \text{CID}_E' \rangle \)

(4.4.16a.iii)

\( \text{FT}^\text{PP}'''' = \langle \text{CID}_E, \text{LT} \rangle \)

(4.4.16a.iv)

\( \text{PP}_1 \leq \text{PP} < \text{PP}_2 \)

(4.4.16a.v)

\( \text{CID}_E' \leq_{\text{CH}} \text{CID}_E \)
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.

\[
\text{ClassHier} \vdash_{bv} \text{Propagate, ClassIdent, ExcAtt}
\]

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) \[ CH \vdash_{bv} \text{false, CID}_E, \text{EA} \]

(4.4.19b) \[ CH \vdash \text{CID}_E, \text{EA} \]

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.

\[
\text{ClassHier} \vdash_{bv} \text{ClassIdent, ExcAtt}
\]

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) \[ CH \vdash \text{CID}_E, \text{CID}_E' \cdot \text{EA} \]

(4.4.20a.i) \[ \text{where } \text{CID}_E \leq_{CH} \text{CID}_E' \]

(4.4.20b) \[ CH \vdash \text{CID}_E, \text{EA} \]

(4.4.20b.i) \[ \text{where } \text{CID}_E \not\leq_{CH} \text{CID}_E' \]
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 thereof) 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 checksum() method code listed in Example 3.4. In order to enhance readability, the table has been divided into six basic blocks. 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 (formalized given by FTA_{pp}.ST) and a local type (formalized 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 checksum(), formalized by M^{ck}, bytecode verifies in the context \( \Gamma^{ck} \) (both specified in Figure 4.6) with the frame type assignment FTA^{ck} (defined in Figure 4.5). Formally speaking we have that

\[
\Gamma^{ck} \vdash_{bv} M^{ck}, FTA^{ck}
\]

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

\[
\begin{align*}
\frac{\text{CH}^{ck} \vdash FTA_0^{ck} \subseteq FT_0^{ck}}{\Gamma^{ck} \vdash_{M^{ck}, FTA^{ck}}} \quad \text{(4.1.28a)}
\end{align*}
\]

\[
\begin{align*}
\frac{\Omega^{ck} \vdash 0, C^{ck}, \emptyset \Rightarrow PPS^{ck}}{\Gamma^{ck} \vdash M^{ck}, FTA^{ck}} \quad \text{(4.2.8b)}
\end{align*}
\]

where

\[
\Delta^{ck} = \left\langle \text{int}, MFR^{ck}, EA^{ck}, ET^{ck} \right\rangle
\]

13 A “basic block” is a code sequence where only the first instruction is a jump target (which includes the initial method invocation jump).
### Figure 4.5: A `cksum()` frame type assignment.
\[ \Gamma^{ck} = \left< \left< \text{CP}^{ck}, \text{Gcd11} \right>, \text{CH}^{ck} \right> \]

\[ \text{CP}^{ck} = \{ 1 \mapsto \text{methodref}(\text{CrCardRd}, \text{methsig}(\text{getIt}, \varepsilon), \text{int}) \}
\]

\[ 2 \mapsto \text{Abort}, \]

\[ 3 \mapsto \text{int} \}

\[ \text{CH}^{ck} = \{ \text{CrCardRd} \mapsto \text{Object}, \text{Gcd11} \mapsto \text{Object}, \]

\[ \text{Abort} \mapsto \text{Exception}, \text{UnsetCrCard} \mapsto \text{Exception} \}

\[ \text{M}^{ck} = \left< \text{MSIG}^{ck}, \text{int}, \text{EA}^{ck}, \text{CA}^{ck} \right> \]

\[ \text{EA}^{ck} = \langle \text{Abort} \rangle \]

\[ \text{MSIG}^{ck} = \text{methsig}(\text{cksum}, \langle \text{CrCardRd} \rangle) \]

\[ \text{CA}^{ck} = \left< \text{MFR}^{ck}, \text{ck}, \text{ET}^{ck} \right> \]

\[ \text{MFR}^{ck} = (2, 5) \]

\[ \text{C}^{ck} = \text{aload}[1] \text{invokevirtual}[1] \text{istore}[2] \text{goto}[+8] \text{pop} \]

\[ \text{new}[2] \text{athrow ldc.w}[3] \text{istore}[3] \text{iload}[2] \text{iload}[3] \]

\[ \text{isub istore}[4] \text{iload}[4] \text{ifle}[+10] \text{iload}[4] \text{istore}[2] \]

\[ \text{goto}[-16] \text{iload}[4] \text{ifne}[+6] \text{iload}[2] \text{return} \]

\[ \text{iload}[2] \text{istore}[4] \text{iload}[3] \text{istore}[2] \text{iload}[4] \]

\[ \text{istore}[3] \text{goto}[-39] \]

\[ \text{ET}^{ck} = \langle \langle 0, 7 \rangle, 10, \text{UnsetCrCard} \rangle \}

\[ \text{FT}^{ck} = (\varepsilon, \langle \text{Gcd11}, \text{CrCardRd}, \perp, \perp, \perp \rangle) \]

\[ \text{PPS}^{ck} = \{ 0, 2, 5, 7, 10, 11, 14, 15, 18, 20, 22, 24, 25, 27, \]

\[ 29, 32, 34, 36, 41, 44, 46, 47, 49, 51, 53, 55, 57, 59 \}

\[ \text{(4.5.14)} \]

Figure 4.6: Context for \text{cksum()} verification.
(4.5.15.ii) \[ \Omega^{ck} = \left\langle \Gamma^{ck}, \Delta^{ck}, \text{FTA}^{ck} \right\rangle \]

and where [A.1.2] denotes the full proof tree given in Appendix A.1. \qed
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 \cite{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 $C_{[PP_1, \ldots, PP_k]}$. We notice:

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

As a consequence we have that from the set of all labels\(^1\) (and 0), given by $\{PP_0, \ldots, PP_k\}$, 
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 
$\{FTA(PP_0), \ldots, FTA(PP_k)\}$ 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 $\{PP_0, \ldots, PP_k\} \in P(PPS_C)$. A frame type set given by $\{FT_{PP_0}, \ldots, FT_{PP_k}\}$ 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 con-
straint set. According to Observation 5.1.1, we can construct a solution $FTA'$ from $FTA_{[PP_1, \ldots, PP_k]}$, 
where $\{PP_1, \ldots, PP_k\}$ is the set of all labels in the method (and 0), in a stepwise manner, starting 
from program point 0. \(\square\)

**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 \cite{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 \cite{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 \cite{20}.) In 
this section, however, we intend to go further in reducing the solution set of needed frame type 
attributions (the certificate) in order to (re)assure bytecode type safety (through type checking). In 
order to suggest a strategy, we shall study an example.

\(^1\)In 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 \((2,3)\), 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 \(CH\).

<table>
<thead>
<tr>
<th>PP</th>
<th>FTA(PP).ST</th>
<th>FTA(PP).LT</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>x \cdot \text{int}</td>
<td>CID \cdot CID_1 \cdot CID_2</td>
<td>pop</td>
</tr>
<tr>
<td>PP +1</td>
<td>x</td>
<td>CID \cdot CID_1 \cdot CID_2</td>
<td>load[1]</td>
</tr>
<tr>
<td>PP +3</td>
<td>x \cdot CID</td>
<td>CID \cdot CID_1 \cdot CID_2</td>
<td>astore[2]</td>
</tr>
<tr>
<td>PP +5</td>
<td>x</td>
<td>CID \cdot CID_1 \cdot CID_2</td>
<td>goto[-4]</td>
</tr>
</tbody>
</table>

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 \(CH \vdash FTA_{pp+6} \subseteq FT_{pp+5}\) (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 \(FT_{PP+1} \subseteq FT_{PP+1}\) and \(FT_{PP+1} \subseteq FT_{PP+5}\). 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” \(FT_{PP+1} \cap FT_{PP+5}\).

<table>
<thead>
<tr>
<th>PP</th>
<th>FTA(PP).ST</th>
<th>FTA(PP).LT</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>x \cdot \text{int}</td>
<td>CID \cdot CID_1 \cdot (CID_1 \cap CID_2)</td>
<td>pop</td>
</tr>
<tr>
<td>PP +1</td>
<td>x</td>
<td>CID \cdot CID_1 \cdot CID_1</td>
<td>load[1]</td>
</tr>
<tr>
<td>PP +3</td>
<td>x \cdot CID</td>
<td>CID \cdot CID_1 \cdot CID_1</td>
<td>astore[2]</td>
</tr>
<tr>
<td>PP +5</td>
<td>x</td>
<td>CID \cdot CID_1 \cdot CID_1</td>
<td>goto[-4]</td>
</tr>
</tbody>
</table>

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.
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

\begin{align*}
CE & \in \text{Cert}_{C, MS, ML} = \text{FrameTypeCert}_{C, MS, ML} \times \text{Labels}_C \\
LS & \in \text{Labels}_C = P(\text{Label}_C) \\
FTC & \in \text{FrameTypeCert}_{C, MS, ML} = \text{PPoints}_C \rightarrow (\text{FrameType}_{MS, ML})^\top \\
\text{Label}_C & = \text{PPoint}_C
\end{align*}

**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
CHAPTER 5. LIGHTWEIGHT VERIFICATION FORMALIZATION

<table>
<thead>
<tr>
<th>Base solution frame-types</th>
<th>Certificate</th>
<th>Safety handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>at backward labels</td>
<td>backward labels (LS)</td>
<td>“saved” (S)</td>
</tr>
<tr>
<td>at forward labels</td>
<td></td>
<td>“pending” (P)</td>
</tr>
<tr>
<td>at “inappropriate” program points</td>
<td>frame types (FTC)</td>
<td></td>
</tr>
</tbody>
</table>

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:

\(^2\)A 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. ANALYSIS AND FORMALIZATION STRATEGY

(5.1.14a) \[ \text{CH} \vdash \text{FTA}(PP') \subseteq \text{FT}_{PP''} \]

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.\footnote{There must be at least two instructions in a method to foster a jump.} We assume an instruction at program point PP, a previous instruction at a program point PP\(^{(-1)}\), and a successor instruction at a program point PP\(^{(+1)}\). Furthermore, a jump instruction at PP\(^{(-i)}\) targeting PP in a forward jump, \(i.e.,\) from PP\(^{(-i)}\) \(<\) PP, and a jump instruction at PP\(^{(+j)}\) targeting PP in a backward jump, \(i.e.,\) from PP \(\leq\) 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.\footnote{There is at least one instruction in any legal method.} Let us chronologically investigate how such a stepwise procedure may progress in the situation of Figure 5.4 where all jumps are shown.
• 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, \( i.e., \ FT^{pp^{(-i)}}_{PP} \), 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 \( FT^{pp^{(-i)}}_{PP} \), 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 \( P(PP) \sqcap FT^{pp^{(-i)}}_{PP} \) (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

\[
(5.1.15a) \quad CH \vdash FTA(PP) \sqsubseteq P(PP)_{pp^{(-i)}}
\]

\[
(5.1.15b) \quad CH \vdash FTA(PP) \sqsubseteq FT^{pp^{(-i)}}_{PP}
\]

and therefore

\[
(5.1.15c) \quad CH \vdash FTA(PP) \sqsubseteq P(PP)_{pp^{(-i)}} \sqcap FT^{pp^{(-i)}}_{PP}, \quad PP > PP^{(-i)}
\]

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, \( P(PP) \sqcap FT^{pp^{(-i)}}_{PP} \) 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 \( PP^{(+i)} \), there is again no branching or confluence in the data flow, as there are no other jump instructions or targeting program points

\(^5\) A non-trivial expected frame type is one which is not \( \top \).

\(^6\) All 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.
5.1. ANALYSIS AND FORMALIZATION STRATEGY

until PP\(^{(i)}\). 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\(^{(i)}\), the lightweight procedure detects a jump instruction which is backward directed towards PP. The jump frame type, FT\(_{PP}^{PP(i)}\), which is the result of applying the jump instruction semantics to the current frame type at PP\(^{(i)}\), 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

\[
(5.1.15d) \quad \text{CH} \vdash S(PP) \sqsubseteq FT_{PP}^{PP(i)}, \ PP < PP^{(i)}
\]

- Finally, from the instruction at PP\(^{(i)}\) 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\(^{(-1)}\), ..., 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 ∈ \(1, ..., k\), which satisfies the following constraint

\[
\text{CH} \vdash \text{FTA}(PP) \sqsubseteq \left( \bigcap_{i=1}^{k} P(PP)_{PP^{(-i)}} \cap FT_{PP}^{PP(-1)} \right)
\]

**Proof.** Follows from Lemma 4.1.38 and the definition of \(\bigcap\) on a complete lattice.

**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\(^{(1)}\), ..., PP\(^{(l)}\) to PP, will satisfy the following constraint

\[
\text{CH} \vdash S(PP) \sqsubseteq \bigcap_{j=1}^{l} FT_{PP}^{PP(j)}
\]

**Proof.** Follows from Lemma 4.1.38 and the definition of \(\bigcap\) on a complete lattice.
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., T, 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'' ≤ PP then PP'' ∈ LS by our certification strategy.
- Whenever FTA(PP) ⊆ \( \left( \bigcap_{i=1}^{k} P(PP)_{pp(-i)} \cap FT_{PP}^{pp(-1)} \right) \), 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 \( \bigcap_{i=1}^{k} P(PP)_{pp(-i)} \cap FT_{PP}^{pp(-1)} \), 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

\[
FTC_0 = \{ PP \mapsto T \mid PP \in PPS_C \}
\]

we have that the following constraint is satisfied for all program points

\[
CH \models FTA(PP) \subseteq \left( FTC(PP) \cap \bigcap_{i=1}^{k} P(PP)_{pp(-i)} \cap FT_{PP}^{pp(-1)} \right)
\]

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 $F_T^{(-1)}_{PP}$. The current frame type is adjusted by the frame type certificate in PP, i.e., $FTC(PP) \sqcap FT_{PP}^{(-1)}$.

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., $FTC(PP) \sqcap P(PP) \sqcap FT_{PP}^{(-1)}$.

Backward target The current frame type, imposed at a backward target PP is saved as $S(PP) = FTC(PP) \sqcap FT_{PP}^{(-1)}$, 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 $S(PP) = FTC(PP) \sqcap P(PP) \sqcap FT_{PP}^{(-1)}$.

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.

\[
P_0 = \{ PP \mapsto \top \mid PP \in PPS_C \}
\]
\[
S_0 = \{ PP \mapsto \bot \mid PP \in PPS_C \}
\]

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 $P' = P \cap \{PP'' \mapsto FT_{PP''}^{PP'}\}$ 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, $S(PP'') \subseteq FT_{PP''}^{PP'}$.

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 $S \cap \{PP \mapsto \bot\}$.

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-analyzed 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 $\{PP \mapsto FT\} \in \text{FrameTypeMap}_{C,MS,ML}^P$ over some singleton domain $\{PP \mid PP \in PPS_C\}$ that

- $P \sqcup \{PP \mapsto FT\}, P \sqcup \{PP \mapsto \top\}$ are well-defined in $\text{FrameTypeMap}_{C,MS,ML}^P$.
- $S \sqcup \{PP \mapsto FT\}, S \cap \{PP \mapsto \bot\}$ are well-defined in $\text{FrameTypeMap}_{C,MS,ML}^S$.

**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 $\langle MS, ML, \text{FrameType}_{MS,ML} \rangle$ defines a lattice (which ensure the existence of “meet” and “join” over the ordered frame type set.)

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

- $P \sqcup \{PP \mapsto FT\}$ 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 $S \cap \{PP \mapsto \bot\}$ 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 $S(PP)$, $P(PP)$, as well as $FTC(PP)$ for a given $PP \in PPoint$ 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 $FT \sqcap FT'$ in side conditions, in order to simplify the formalization presentations. We notice that a correct but more cumbersome approach, would be to specify $CH \vdash FT \sqcap FT'$ 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

$$CST \in \text{CodeStat}_{C,MS,ML} = \text{PPoint}_C \times (\text{FrameType}_{MS,ML})^\top$$

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

$$\Omega_{light} \in \text{LightContext}_{C,MS,ML} = \text{StdContext} \times \text{MethContext}_{C,MS,ML} \times \text{Cert}_{C,MS,ML}$$

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 $\emptyset$ 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

\[ \text{StdContext} \vdash_{\text{lbv}} \text{Method, Cert} \]

where “\(\Gamma \vdash_{\text{lbv}} M, CE\)” reads: the method \(M\) lightweight verifies with the certificate \(CE\) in the standard verification context \(\Gamma\).

\[
\Omega_{\text{light}} \vdash \text{CST}_0, C, \theta, \langle \text{P}_0, \text{S}_0 \rangle \overset{\text{tsafe}}{\Rightarrow} \text{PPS}, \langle \text{P}_0, \text{S} \rangle
\]

\(\Gamma \vdash_{\text{lbv}} M : CE\)

(5.2.5a)

where \(\Omega_{\text{light}} = \langle \Gamma, \Delta, CE \rangle\)

(5.2.5a.i) \(CE = \langle \text{FTC}, \text{LS} \rangle\)

(5.2.5a.ii) \(\text{CST}_0 = \langle 0, \text{FT}_0 \rangle\)

(5.2.5a.iv) \(\text{P}_0 = \{ \text{PP} \mapsto \top \mid \text{PP} \in \text{PPS}_C \}\)

(5.2.5a.v) \(\text{S}_0 = \{ \text{PP} \mapsto \bot \mid \text{PP} \in \text{PPS}_C \}\)

(5.2.5a.vi) 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 \(\text{PP}\) is abbreviated to \(\text{FT}_1^{\text{PP}}\).

- The current frame type is generally set to \(\text{FTC}(\text{PP}) \sqcap \text{P}(\text{PP}) \sqcap \text{FT}_\text{PP}\) regardless of the status of \(\text{PP}\), as we observe that \(\text{P}(\text{PP}) = \top\) for non-targets or backward targets. Thus, \(\text{P}(\text{PP})\) does not contribute to the frame type value in those cases.

- The modification of the pending map \(\text{P}\) is generally set to \(\text{P} \sqcup \{ \text{PP} \mapsto \top \}\) regardless of the status of \(\text{PP}\), as we observe that \(\text{P}(\text{PP}) = \top\) for non-targets or backward targets.

- The modification of the saved map \(\text{S}\) is generally set to \(\text{S} \sqcup \{ \text{PP} \mapsto \text{FT}_1^{\text{PP}} \}\) regardless of the status of \(\text{PP}\). Even though the extension of \(\text{S}\) 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.
### Definition 5.2.6 (Instruction Sequence Lightweight Verification)

A code sequence, lightweight verification judgment has the signature

\[
\text{LightContext} \vdash_{\text{lvv}} \text{CodeStat, CodeSeq, PPoints, DelayConstr} \Rightarrow_{\text{itsafe}} \text{PPoints, DelayConstr}
\]

where “\(\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle, CS, PPS, DC \Rightarrow_{\text{itsafe}} PPS', DC'\)” reads: the code sequence \(CS\), starting at the program point \(PP\) with the frame type \(FT_{PP}\), lightweight verifies in the context \(\Omega_{\text{light}}\) 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 \(DC'\).

\[
\begin{align*}
\Omega_{\text{light}} & \vdash \text{CST}', 1, \langle P, S \rangle \Rightarrow_{\text{itsafe}} \langle PP', \top \rangle, P_0 \\
\Omega_{\text{light}} & \vdash \text{CST}, 1, \text{PPS}, \langle P, S \rangle \Rightarrow_{\text{itsafe}} \text{PPS}, \langle P, S \rangle \\
\text{where} & \quad \Omega_{\text{light}} = \langle \omega, \omega, \text{CE} \rangle \\
\text{CE} & = \langle \text{FTC}, \omega \rangle \\
\text{CST} & = \langle PP, FT_{PP} \rangle \\
\text{CST}' & = \langle PP, \text{FTC}(PP) \cap P(PP) \cap FT_{PP} \rangle \\
P_0 & = P \cup \{PP \mapsto \top\} \\
\text{same side condition as in (4.2.8a.i).}
\end{align*}
\]

\[
\begin{align*}
\Omega_{\text{light}} & \vdash \text{CST}', 1, \langle P', S' \rangle \Rightarrow_{\text{itsafe}} \text{CST}'' , P'' \\
\Omega_{\text{light}} & \vdash \text{CST}'' , C, \text{PPS}', \langle P'', S' \rangle \Rightarrow_{\text{itsafe}} \text{PPS}'' , \langle P''', S''' \rangle \\
\Omega_{\text{light}} & \vdash \text{CST}, 1 \cdot C, \text{PPS}, \langle P, S \rangle \Rightarrow_{\text{itsafe}} \text{PPS}'' , \langle P''', S''' \rangle
\end{align*}
\]
5.3. INSTRUCTION VERIFICATION

(5.2.6b.i) where \( \Omega_{\text{light}} = \langle \_ , \_ , \_ , \_ , CE \rangle \)

(5.2.6b.ii) \( CE = \langle FTC, \_ \rangle \)

(5.2.6b.iii) \( CST = \langle PP, FT_{PP} \rangle \)

(5.2.6b.iv) \( CST' = \langle PP, FT_{PP}^{1} \rangle \)

(5.2.6b.vi) \( FT_{PP}^{1} = FTC(PP) \cap P(PP) \cap FT_{PP} \)

(5.2.6b.vii) \( P' = P \cup \{ PP \mapsto T \} \)

(5.2.6b.viii) \( S' = S \cup \{ PP \mapsto FT_{PP}^{1} \} \)

(5.2.6b.ix) same side conditions as in (4.2.8b.ii) through (4.2.8b.iii).

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 \( S \cup \{ PP \mapsto FT \} \) defines an extension of \( S \) and the formula \( P \cup \{ PP \mapsto T \} \), 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.

**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 \( FTC(PP') \cap FT_{PP}^{1} \cap FT_{PP} \cap P(PP') \) 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
The formal difference is that the FTA component of the code context $\Omega$ has been "reduced" to CE in $\Omega_{\text{light}}$. 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 $\text{FTA}(PP)$). 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.

![Figure 5.9: Delayed branching constraints.](image)

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

$$
\text{LightContext} \vdash_{\text{lbv}} \text{CodeStat} : \text{Ins}, \text{DelayConstr} \xrightarrow{\text{ltsafe}} \text{CodeStat}, \text{Pending}
$$

where "$\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle : I, \langle P, S \rangle \xrightarrow{\text{ltsafe}} \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 $\Omega_{\text{light}}$, 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)

$$
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle : I, \langle P, S \rangle \xrightarrow{\text{ltsafe}} \langle PP', FT_{PP}^{PP'} \rangle, P
$$

(5.3.2a.i) where $\Omega_{\text{light}} = \langle I, \Delta, \_ \rangle$

(5.3.2a.ii) $FT_{PP} = \langle ST, LT \rangle$

(5.3.2a.iii) 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) \( \Omega_{\text{light}} = (\Gamma, \Delta, \_\_ \_\_) \)

(5.3.3a.ii) \( \text{FT}_{PP} = \langle ST, LT \rangle \)

(5.3.3a.iii) \text{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) \[ \Theta_{\text{light}} \vdash PP, ES, \langle P, S \rangle \xrightarrow{\text{tsafe}} P' \]

(5.3.4a.i) \( \Omega_{\text{light}} = (\Gamma, \Delta, \_\_ \_\_ \_) \)

(5.3.4a.ii) \( \Theta_{\text{light}} = (\Gamma, \Delta, LT, \text{false}) \)

(5.3.4a.iii) \text{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) \[ \Theta_{\text{light}} \vdash PP, ES, \langle P, S \rangle \xrightarrow{\text{tsafe}} P' \]

(5.3.5a.i) \( \Omega_{\text{light}} = (\Gamma, \Delta, \_\_ \_\_ \_) \)

(5.3.5a.ii) \( \Theta_{\text{light}} = (\Gamma, \Delta, LT, \text{false}) \)

(5.3.5a.iii) \text{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.
CHAPTER 5. LIGHTWEIGHT VERIFICATION FORMALIZATION

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.

\[
\text{CH} \vdash T \subseteq \tau
\]

\[
\Theta_{\text{light}} \vdash PP, ES, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P'
\]

\[
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle : I, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} \langle PP', FT_{PP'} \rangle , P'
\]

(5.3.6a)

where \( \Omega_{\text{light}} = \langle \Gamma, \Delta, \_ \rangle \)

(5.3.6a.i)

\( \Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle \)

(5.3.6a.ii)

\( \text{same side conditions as in (4.3.16a.iv) through (4.3.16a.ix)).} \)

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.

\[
\text{CH} \vdash T_1 \subseteq \tau_1
\]

\[
\text{CH} \vdash T_j \subseteq \tau_j
\]

\[
\Theta_{\text{light}} \vdash PP, ES, ET, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P'
\]

\[
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} \rangle : I, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} \langle PP', FT_{PP'} \rangle , P'
\]

(5.3.7a)

where \( \Omega_{\text{light}} = \langle \Gamma, \Delta, \_ \rangle \)

(5.3.7a.i)

\( \Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle \)

(5.3.7a.ii)

\( \text{same side conditions as in (4.3.17a.iv) through (4.3.17a.ix)).} \)

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.

\[
5.3.8a \quad \text{CH} \vdash S(PP') \subseteq FT_{PP'} \\
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} : 1, \langle P, S \rangle \xrightarrow{\text{ltsafe}} \langle PP', FT_{PP'} \rangle, P \rangle \\
\text{where} \quad PP' \subseteq PP
\]

\[
5.3.8a.i \quad \text{same side conditions as in (4.3.20a.iii) through (4.3.20a.vii)).}
\]

\[
5.3.8b \quad \text{CH} \vdash S(PP') \subseteq FT_{PP'} \\
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} : 1, \langle P, S \rangle \xrightarrow{\text{ltsafe}} \langle PP', FT_{PP'} \rangle, P \rangle \\
\text{where} \quad PP' > PP
\]

\[
5.3.8b.i \quad \text{same side conditions as in (4.3.20a.iii) through (4.3.20a.vii)).}
\]

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.

\[
5.3.9a \quad \text{CH} \vdash S(PP') \subseteq FT_{PP} \\
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} : \text{goto}[n], \langle P, S \rangle \xrightarrow{\text{ltsafe}} \langle PP', \top \rangle, P \rangle \\
\text{where} \quad PP' \subseteq PP
\]

\[
5.3.9a.i \quad \text{same side conditions as in (4.3.22a.iii) through (4.3.22a.v)).}
\]

\[
5.3.9b \quad \text{CH} \vdash S(PP') \subseteq FT_{PP} \\
\Omega_{\text{light}} \vdash \langle PP, FT_{PP} : \text{goto}[n], \langle P, S \rangle \xrightarrow{\text{ltsafe}} \langle PP', \top \rangle, P \rangle \\
\text{where} \quad PP' > PP
\]

\[
5.3.9b.i \quad \text{same side conditions as in (4.3.22a.iii) through (4.3.22a.v)).}
\]
The lightweight verification rule for the \texttt{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, \texttt{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.

\begin{align*}
\Theta_{\text{light}} & \vdash \text{PP}, \text{CID}_{E}, \text{ET}, \langle P, S \rangle_{\text{safe}} \to P' \\
\Omega_{\text{light}} & \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle: \text{throw}, \langle P, S \rangle_{\text{safe}} \to \langle \text{PP}' , \top \rangle, P'
\end{align*}

\begin{enumerate}
\item[(5.3.10a.i)] where $\Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$
\item[(5.3.10a.ii)] $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$
\item[(5.3.10a.iii)] \textit{same side conditions as in (4.3.24a.iv) through (4.3.24a.viii)}
\end{enumerate}

\begin{align*}
\text{(5.3.10b)} & \quad \text{same rule as in (5.3.10a).}
\end{align*}

\begin{enumerate}
\item[(5.3.10b.i)] where $\Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$
\item[(5.3.10b.ii)] $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{true} \rangle$
\item[(5.3.10b.iii)] \textit{same side conditions as in (4.3.24b.iv) through (4.3.24b.viii)}
\end{enumerate}

\begin{align*}
\text{(5.3.10c)} & \quad \text{same rule as in (5.3.10a).}
\end{align*}

\begin{enumerate}
\item[(5.3.10c.i)] where $\Omega_{\text{light}} = \langle \Gamma, \Delta, \text{CE} \rangle$
\item[(5.3.10c.ii)] $\Theta_{\text{light}} = \langle \Gamma, \Delta, \text{LT}, \text{false} \rangle$
\item[(5.3.10c.iii)] \textit{same side conditions as in (4.3.24c.iv) through (4.3.24c.vii)}
\end{enumerate}

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.

\begin{align*}
\Omega_{\text{light}} & \vdash \langle \text{PP}, \text{FT}_{\text{PP}} \rangle: \text{return}, \langle P, S \rangle_{\text{safe}} \to \langle \text{PP}' , \top \rangle, P
\end{align*}
5.4. EXCEPTION VERIFICATION

where \( \Omega_{\text{light}} = \langle \cdot \Delta \cdot \rangle \)

*same side conditions as in* (4.3.25a.ii) *through* (4.3.25a.iii).

"same rule as in" (5.3.11a).

where \( \Omega_{\text{light}} = \langle \cdot \Delta \cdot \rangle \)

*same side conditions as in* (4.3.25b.iii) *through* (4.3.25b.iv).

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 \( \{pp'' \mapsto \text{FT}_{pp''} \} \) 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 alteration 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 $\Theta$ 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.
5.4. EXCEPTION VERIFICATION

<table>
<thead>
<tr>
<th>Verifying rule</th>
<th>Lightweight rules</th>
<th>Catch</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4.4.12a)</td>
<td>(5.4.5a)</td>
<td>no</td>
</tr>
<tr>
<td>(4.4.12b)</td>
<td>(5.4.5b)</td>
<td>no</td>
</tr>
<tr>
<td>(4.4.12c)</td>
<td>(5.4.5c)</td>
<td>no</td>
</tr>
<tr>
<td>(4.4.12d)</td>
<td>(5.4.5d)</td>
<td>no</td>
</tr>
<tr>
<td>(4.4.12e)</td>
<td>(5.4.5e)</td>
<td>no</td>
</tr>
<tr>
<td>(4.4.14a)</td>
<td>(5.4.6a), (5.4.6b)</td>
<td>definite</td>
</tr>
<tr>
<td>(4.4.16a)</td>
<td>(5.4.7a), (5.4.7b)</td>
<td>potential</td>
</tr>
</tbody>
</table>

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.

\[ \text{ExcContext} \vdash_{bv} \text{PP}, \text{Excs}, \text{ExcHandlers}, \text{DelayConstr} \vdash_{\text{itsafe}} \text{Pending} \]

where “\( \Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{EHS}, \langle \text{P}, \text{S} \rangle \vdash_{\text{itsafe}} \text{P} \)” reads: the set of exceptions \( \text{ES} \), which can be raised at a program point \( \text{PP} \), lightweight verifies on a list of exception handlers \( \text{EHS} \) on a delayed constraint set \( \langle \text{P}, \text{S} \rangle \), within the lightweight context \( \Theta_{\text{light}} \), with an accumulated pending set \( \text{P} \).

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.

\[
(5.4.5a) \quad \Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E, \epsilon, \langle \text{P}, \text{S} \rangle \vdash_{\text{itsafe}} \text{P}
\]
where \( \Theta_{\text{light}} = \langle \langle \_, \text{CH} \rangle, \langle \_, \text{EA}, \epsilon \rangle, \_, \text{PR} \rangle \)

\[
\text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA}
\]

(5.4.5a.i)

\[
\Theta_{\text{light}} \vdash \text{PP}, \text{ES}, \text{EHS}, \langle P, S \rangle \xrightarrow{\text{lt-safe}} P'
\]

(5.4.5b)

\[
\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \epsilon, \langle P, S \rangle \xrightarrow{\text{lt-safe}} P'
\]

(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)

\[
\Theta_{\text{light}} \vdash \text{PP}, \epsilon, \text{EHS}, \langle P, S \rangle \xrightarrow{\text{lt-safe}} P
\]

(5.4.5d)

\[
\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \langle P, S \rangle \xrightarrow{\text{lt-safe}} P'
\]

(5.4.5d.i)

*same side conditions as in (4.4.12d.i) through (4.4.12d.ii).*

(5.4.5e)

\[
\Theta_{\text{light}} \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \langle P, S \rangle \xrightarrow{\text{lt-safe}} P'
\]

(5.4.5e.i)

where \( \Theta_{\text{light}} = \langle \langle \_, \text{CH} \rangle, \_, \_, \text{PR} \rangle \)

(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 \( \text{CID}_E \) 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.

\[
CH \vdash S(PP') \subseteq FT_{pp}^{pp'}
\]

\[
\Theta_{\text{light}} \vdash PP, ES, ET, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P'
\]

\[
\Theta_{\text{light}} \vdash PP, CID_E \cdot ES, EH \cdot EHS, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P''
\]

(5.4.6a)

(5.4.6a.i) \hspace{1cm} \text{where } PP'' \leq PP

(5.4.6a.ii) \hspace{1cm} \Theta_{\text{light}} = \langle \langle -, CH \rangle, \langle -, -, ET \rangle, LT, \_ \rangle

(5.4.6a.iii) \hspace{1cm} \text{same side conditions as in (4.4.14a.iii) through (4.4.14a.vi).}

(5.4.6b)

\[
\Theta_{\text{light}} \vdash PP, ES, ET, \langle P', S \rangle \overset{\text{itsafe}}{\rightarrow} P''
\]

\[
\Theta_{\text{light}} \vdash PP, CID_E \cdot ES, EH \cdot EHS, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P''
\]

(5.4.6b.i)

\[
\Theta_{\text{light}} = \langle \langle -, CH \rangle, \langle -, -, ET \rangle, LT, \_ \rangle
\]

(5.4.6b.ii) \hspace{1cm} \text{where } PP'' > PP

(5.4.6b.iii) \hspace{1cm} P' = P \cap \{PP'' \mapsto FT_{pp''}^{PP''}\}

(5.4.6b.iv) \hspace{1cm} \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.

\[
CH \vdash S(PP') \subseteq FT_{pp}^{pp'}
\]

\[
\Theta_{\text{light}} \vdash PP, CID_E \cdot ES, EHS, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P'
\]

\[
\Theta_{\text{light}} \vdash PP, CID_E \cdot ES, EH \cdot EHS, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P''
\]

(5.4.7a)

(5.4.7a.i) \hspace{1cm} \text{where } PP'' \leq PP

(5.4.7a.ii) \hspace{1cm} \Theta_{\text{light}} = \langle \langle -, CH \rangle, \langle -, -, ET \rangle, LT, \_ \rangle

(5.4.7a.iii) \hspace{1cm} \text{same side conditions as in (4.4.16a.iii) through (4.4.16a.vi).}

(5.4.7b)

\[
\Theta_{\text{light}} \vdash PP, CID_E \cdot ES, EHS, \langle P', S \rangle \overset{\text{itsafe}}{\rightarrow} P''
\]

\[
\Theta_{\text{light}} \vdash PP, CID_E \cdot ES, EH \cdot EHS, \langle P, S \rangle \overset{\text{itsafe}}{\rightarrow} P''
\]
where $\text{PP}'' > \text{PP}$

\[
\Theta_{\text{light}} = \langle \_, \text{CH}, \_, \text{LT}, \_ \rangle
\]

$P' = P \cap \{\text{PP}'' \rightarrow \text{FT}_{\text{PP}''}\}$

**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 $\Gamma^{ck}$, i.e., $\Gamma^{ck} \vdash_{\text{lbv}} M^{ck}, CE^{ck}$, with the certificate:

\[
\begin{align*}
(5.5.1a) & \quad CE^{ck} = \langle \text{FTC}^{ck}, \nu L^{ck} \rangle \\
(5.5.1b) & \quad FTC^{ck} = \emptyset \\
(5.5.1c) & \quad L^{ck} = \{20\}
\end{align*}
\]

**Proof.** Assume $\Gamma^{ck}, M^{ck},$ and $CE^{ck}$ as in the Proposition. Then the lightweight rules of this chapter can be used to construct the proof

\[
\begin{align*}
A.2.1 \\
\Omega^{ck}_{\text{light}} \vdash \langle \emptyset, \text{FT}^{ck}_0 \rangle, C^{ck}_0, 0, \langle P^{ck}_0, S^{ck}_0 \rangle \xrightarrow{\text{litsafe}} \text{PPS}^{ck}, \langle P^{ck}_0, S^{ck}_{20} \rangle \\
\Gamma^{ck} \vdash M^{ck}, CE^{ck}
\end{align*}
\]

where

\[
\begin{align*}
(5.5.2.i) & \quad \Omega^{ck}_{\text{light}} = \langle \Gamma^{ck}, \Delta^{ck}, CE^{ck} \rangle \\
(5.5.2.ii) & \quad \Delta^{ck} = \langle \text{int}, \text{MFR}^{ck}, \text{EA}^{ck}, \text{ET}^{ck} \rangle \\
(5.5.2.iii) & \quad P^{ck}_0 = \{ \text{PP} \rightarrow \top \mid \text{PP} \in \text{PPS}^c \} \\
(5.5.2.iv) & \quad S^{ck}_0 = \{ \text{PP} \rightarrow \bot \mid \text{PP} \in \text{PPS}^c \} \\
(5.5.2.v) & \quad S^{ck}_{20} = \{ \text{PP} \rightarrow \bot \mid \text{PP} \in \text{PPS}^c \} \sqcup \{20 \rightarrow \text{FTA}(20)\}
\end{align*}
\]

and where $\square$ 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 bytecode, 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 $\Gamma$ and method $M$ the following statements are equivalent:

1. There exists a frame type assignment $FTA$ such that $\Gamma \vdash_{by} M, FTA$ 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.

**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

\[(6.1.2a) \quad CH \vdash S(PP) \sqsubseteq FT_{PP}^{(j)} , \quad PP \leq PP^{(j)} \]

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 cksu(), 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

\[(6.1.2b) \quad CH \vdash \bigcap_{i=1}^{k} P(PP)_{pp(-i)} \cap FT_{PP}^{(i)} , \quad PP > PP^{(-i)} \]

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 \(T\). 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 cksu(), 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 \( \text{FTA}' \) 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

\[
\text{CH} \models \text{FTA}'(PP) \sqsubseteq \bigcap_{i=1}^{k} P(PP)_{pp(-i)} \cap FT^{p(-1)}_{pp}, \quad PP > PP^{(-1)}
\]

For the solution \( \text{FTA}' \) in Figure 6.1, this means that the frame type component of the certificate must be given by

\[
\text{FTC}' = \{ 0 \mapsto \text{FTA}'(0), 5 \mapsto \text{FTA}'(5), 10 \mapsto \text{FTA}'(10), 32 \mapsto \text{FTA}'(32), 41 \mapsto \text{FTA}'(41), 44 \mapsto \text{FTA}'(44), 49 \mapsto \text{FTA}'(49), 53 \mapsto \text{FTA}'(53) \}
\]

as the constraint

\[
\text{CH} \models \text{FTA}'(PP) \sqsubseteq \left( \bigcap_{i=1}^{k_{PP}} P(PP)_{pp(-i)} \cap FT^{p(-1)}_{pp} \right)
\]

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 \( FT^{0(-1)}_{0} \) is defined as \( FT_{0} \).

In comparison, we have that for the solution \( \text{FTA} \) in Figure 4.5 of Example 4.5, the frame type component, deduced by the same certificate principles, becomes

\[
\text{FTC} = \emptyset
\]

We notice, that the deduced label set \( LS \) for the code associated with the solution \( \text{FTA}' \) in Example 6.1, is the same as for \( \text{FTA} \) in Example 6.1, as the code sequence of the \( \text{cksum}() \) 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
## Chapter 6. Lightweight Certification Formalization

**Table 6.1:** A “smaller” `cksum()` frame type assignment.

<table>
<thead>
<tr>
<th>PP</th>
<th><code>FTA_{pp.ST}</code></th>
<th><code>FTA_{pp.LT} (this \cdot ccnum \cdot x \cdot y \cdot z)</code></th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>e</code></td>
<td>⊥·CrCardRd·⊥·⊥·⊥</td>
<td><code>aload[1]</code></td>
</tr>
<tr>
<td>2</td>
<td>CrCardRd</td>
<td>⊥·CrCardRd·⊥·⊥·⊥</td>
<td><code>invokevirtual[1]</code></td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>⊥·⊥·⊥·⊥·⊥</td>
<td><code>istore[2]</code></td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>⊥·⊥·⊥·⊥·⊥</td>
<td><code>goto[+8]</code></td>
</tr>
<tr>
<td>10</td>
<td>⊥</td>
<td>⊥·⊥·⊥·⊥·⊥</td>
<td><code>pop</code></td>
</tr>
<tr>
<td>11</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·⊥·⊥</td>
<td><code>new[2]</code></td>
</tr>
<tr>
<td>14</td>
<td>Abort</td>
<td>⊥·⊥·⊥·⊥·⊥</td>
<td><code>athrow</code></td>
</tr>
<tr>
<td>15</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·int·⊥</td>
<td><code>ldc[w][3]</code></td>
</tr>
<tr>
<td>18</td>
<td>int</td>
<td>⊥·⊥·⊥·int·⊥</td>
<td><code>istore[3]</code></td>
</tr>
<tr>
<td>20</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·int·int·⊥</td>
<td><code>iload[2]</code></td>
</tr>
<tr>
<td>22</td>
<td>int</td>
<td>⊥·⊥·⊥·int·int·⊥</td>
<td><code>iload[3]</code></td>
</tr>
<tr>
<td>24</td>
<td>int·int</td>
<td>⊥·⊥·⊥·int·int·⊥</td>
<td><code>isub</code></td>
</tr>
<tr>
<td>25</td>
<td>int</td>
<td>⊥·⊥·⊥·int·int·⊥</td>
<td><code>istore[4]</code></td>
</tr>
<tr>
<td>27</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·int·int·int</td>
<td><code>iload[4]</code></td>
</tr>
<tr>
<td>29</td>
<td>int</td>
<td>⊥·⊥·⊥·int·int·int</td>
<td><code>ifle[+10]</code></td>
</tr>
<tr>
<td>32</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>iload[4]</code></td>
</tr>
<tr>
<td>34</td>
<td>int</td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>istore[4]</code></td>
</tr>
<tr>
<td>36</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>goto[+5]</code></td>
</tr>
<tr>
<td>39</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·int·int·int</td>
<td><code>iload[4]</code></td>
</tr>
<tr>
<td>41</td>
<td>int</td>
<td>⊥·⊥·⊥·int·int·int</td>
<td><code>ifne[+6]</code></td>
</tr>
<tr>
<td>44</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>iload[2]</code></td>
</tr>
<tr>
<td>46</td>
<td>int</td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>ireturn</code></td>
</tr>
<tr>
<td>47</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·int·int·⊥</td>
<td><code>iload[2]</code></td>
</tr>
<tr>
<td>49</td>
<td>int</td>
<td>⊥·⊥·⊥·⊥·int·⊥</td>
<td><code>istore[4]</code></td>
</tr>
<tr>
<td>51</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>iload[3]</code></td>
</tr>
<tr>
<td>53</td>
<td>int</td>
<td>⊥·⊥·⊥·⊥·⊥·int</td>
<td><code>istore[2]</code></td>
</tr>
<tr>
<td>55</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·⊥·int·int</td>
<td><code>iload[4]</code></td>
</tr>
<tr>
<td>57</td>
<td>int</td>
<td>⊥·⊥·⊥·int·⊥·⊥</td>
<td><code>istore[3]</code></td>
</tr>
<tr>
<td>59</td>
<td><code>e</code></td>
<td>⊥·⊥·⊥·int·int·⊥</td>
<td><code>goto[−39]</code></td>
</tr>
</tbody>
</table>

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 \texttt{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 \( \text{FTA}' \) to the method’s constraint set, such that \( \text{FTA}'(PP) \subseteq \text{FTA}(PP) \) for all \( PP \in \text{PPS}_C \). 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 \( PP'' \subseteq PP \) then \( PP'' \in \text{LS} \) by our certification strategy.

2. Whenever \( CH \vdash \text{FTA}(PP) \sqsubseteq \left( \prod_{i=1}^{k} P(PP)_{pp(-i)} \cap \text{FT}_{pp}^{(-i)} \right) \), we have that \( \text{FTC} \) 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 \( \text{FTC} \), appropriately, whereas the latter constraints serve to extend the backward label set, syntactically given by \( \text{LS} \). 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 FTA(PP) is less defined that the current frame type CH ⊨ \bigcap_{i=1}^{k} P(PP)_{pp(-i)} \cap FT^{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) \[ FTC_0 = \{\text{PP} \mapsto \top \mid \text{PP} \in PPS_C\} \]

### 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 PPS_C\}.
- We introduce the following abbreviations for each of the three formal verification semantics in this thesis. Here specified by their principal judgement.

  (BV) \[ \Gamma \vdash_{bv} M, FTA \quad \text{(Standard bytecode verification)} \]
  (LBV) \[ \Gamma \vdash_{lbv} M, CE \quad \text{(Lightweight bytecode verification)} \]
  (LBC) \[ \Gamma \vdash_{lbc} M, FTA, CE \quad \text{(Lightweight bytecode certification)} \]

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) \[ PCE \in PCert_{C,MS,ML} = \text{Pending}_{C,MS,ML} \times Cert_{C,MS,ML} \]
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

$$\text{StdContext} \vdash_{lbc} \text{Method, FrameTypeApprox, Cert}$$

where “$\Gamma \vdash_{lbc} M, FTA, CE$” reads: the method code $M$ bytecode-certifies with a frame type assignment $FTA$ in the verification context $\Gamma$, with an accumulating certificate $CE$.

$$CH \vdash FTA(0) = FT_0$$
$$\Omega \vdash 0, C, \emptyset, PCE_0 \text{ certify } PPS, PCE$$

$\Gamma \vdash_{lbc} M, FTA, CE$

(6.2.4a)

where $FTC_0 = \{ PP \mapsto T \mid PP \in PPS_C \}$

(6.2.4a.i)

$P_0 = \{ PP \mapsto T \mid PP \in PPS_C \}$

(6.2.4a.ii)
CHAPTER 6. LIGHTWEIGHT CERTIFICATION FORMALIZATION

(6.2.4a.iii) \[ \text{PCE}_0 = \langle P_0, \langle FTC_0, 0 \rangle \rangle \]
(6.2.4a.iv) \[ \text{PCE} = \langle P_0, \text{CE} \rangle \]
(6.2.4a.v) \[ \text{same side conditions as in (4.2.7a.i) through (4.2.7a.x).} \]

\[
\begin{align*}
\text{CH} & \vdash \text{FTA}(0) \sqsubseteq FT_0 \\
\Omega & \vdash 0, C, \emptyset, \text{PCE}_0 \text{ certify } \Rightarrow \text{PPS}, \text{PCE}
\end{align*}
\]
(6.2.4b)

(6.2.4b.i) \[ \text{where FTC}_1 = \text{FTC}_0 \cap \{0 \mapsto \text{FTA}(0)\} \]
(6.2.4b.ii) \[ \text{FTC}_0 = \{ PP \mapsto T \mid PP \in \text{PPS}_C \} \]
(6.2.4b.iii) \[ P_0 = \{ PP \mapsto T \mid PP \in \text{PPS}_C \} \]
(6.2.4b.iv) \[ \text{PCE}_0 = \langle P_0, \langle \text{FTC}_1, 0 \rangle \rangle \]
(6.2.4b.v) \[ \text{PCE} = \langle P_0, \text{CE} \rangle \]
(6.2.4b.vi) \[ \text{same side conditions as in (4.2.7a.i) through (4.2.7a.x).} \]

\textbf{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).

\textbf{Lemma 6.2.6 (BV-LBC and LBC-LBV equivalences).} Within the verification context \( \Gamma \), we have that if a method \( M \) standard verifies successfully with a frametype approximation \( \text{FTA} \), then there exists a uniquely given certicate \( \text{CE} \) by which the method certifies, and vice versa.

\begin{equation}
\Gamma \vdash_{\text{bv}} M, \text{FTA} \Leftrightarrow \exists! \text{CE} : \Gamma \vdash_{\text{lbc}} M, \text{FTA}, \text{CE}
\end{equation}

(6.2.6a)

Furthermore we have that within the verification context \( \Gamma \), if a method \( M \) can be lightweight verified with a given certicate \( \text{CE} \), then \( \text{CE} \) is unique in assisting \( M \) to be certified with some frame type assignment \( \text{FTA} \), and vice versa.

\begin{equation}
\Gamma \vdash_{\text{lbc}} M, \text{CE} \Leftrightarrow \exists! \text{FTA} : \Gamma \vdash_{\text{lbc}} M, \text{FTA}, \text{CE}
\end{equation}

(6.2.6b)

\textbf{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.

\textbf{Case BV \( \Rightarrow \) LBC.} Assume a BV proof tree with (4.2.7a) at the base and subproofs for \( \text{CH} \vdash \text{FTA}(0) \sqsubseteq FT_0 \) and \( \Omega \vdash 0, C, \emptyset \text{ safe } \Rightarrow \text{PPS} \). We can identify the former as either a subproof for frame type equality \( \text{CH} \vdash \text{FTA}(0) = FT_0 \) or frame type reduction \( \text{CH} \vdash \text{FTA}(0) \sqsubseteq FT_0 \).

From Lemma 6.2.14 we will have a subproof for \( \Omega \vdash 0, C, \emptyset, \text{PCE}_0 \text{ certify } \Rightarrow \text{PPS}, \text{PCE} \) 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 \( \exists! \text{CE} : \Gamma \vdash_{\text{lbc}} M, \text{FTA}, \text{CE} \) with either (6.2.4a) or (6.2.4b) as the base rule.
6.2. BYTECODE CERTIFICATION

Case BV $\Leftrightarrow$ 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 $CH \vdash FTA(0) \sqsubseteq FT_0$ and $\Omega \vdash 0, c, 0 \stackrel{\text{tsafe}}{\Rightarrow} PPS$ 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 $\Omega_{\text{light}} \vdash \text{CST}_0, c, \emptyset, \langle P_0, S_0 \rangle \stackrel{\text{tsafe}}{\Rightarrow} PPS, \langle P_0, S \rangle$ From Lemma 6.2.14 we will have a proof of $\Omega \vdash 0, c, \emptyset, \text{PCE}_0 \stackrel{\text{ certify}}{\Rightarrow} PPS, \text{PCE}$ where one of the following two situations hold: either $CH \vdash FTA(0) = FT_0$ is provable and the side conditions of (6.2.4a) hold, or $CH \vdash FTA(0) \sqsubseteq FT_0$ 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 $\Leftrightarrow$ LBC. Assume an LBC proof tree with either (6.2.4a) or (6.2.4b) at the base. With $\Omega_{\text{light}}, \text{CST}_0, \text{and} S_0$ defined as in the side conditions of Definition 5.2.5 we then by Lemma 6.2.14 will have a subproof for $\Omega_{\text{light}} \vdash \text{CST}_0, c, \emptyset, \langle P_0, S_0 \rangle \stackrel{\text{tsafe}}{\Rightarrow} PPS, \langle P_0, S \rangle$ which means that we have a proof of $\Gamma \vdash_{\text{lbv}} M, CE$.

The four cases combine to prove the two equivalences.

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 a “pre-delayed constraint” for the method is formally specified by

$$PCT \in \text{PreConstr}_{C, MS, ML} = \text{Pending}_{C, MS, ML} \times \text{Labels}_C$$

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 map$^1$ 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

$$\text{CodeContext} \vdash_{\text{lbc}} \text{PPoint, CodeSeq, PPoints, PCert} \stackrel{\text{ certify}}{\Rightarrow} \text{PPoints, PCert}$$

$^1$Both 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 \( \Omega \) with an accumulated set of certified program points \( PPS' \), and an accumulated, pending-certificate \( PCE' \).

\[
\begin{align*}
\Omega \vdash PP, I, PCT &\xrightarrow{\text{certify}} PP', T, PCT_0 \\
\Omega \vdash PP, I, PPS, PCE &\xrightarrow{\text{certify}} PPS', PCE'
\end{align*}
\]

(6.2.9a) \( PCE = \langle P, CE \rangle \)

(6.2.9a.i) \( PCE' = \langle P_0, CE \rangle \)

(6.2.9a.ii) \( CE = \langle \_ , LS \rangle \)

(6.2.9a.iii) \( PCT = \langle P, LS \rangle \)

(6.2.9a.iv) \( PCT_0 = \langle P_0, LS \rangle \)

(6.2.9a.v) \( P_0 = P \cup \{ PP \rightarrow T \} \)

(6.2.9a.vi) \( \text{same side condition as in (4.2.8a.i)} \)

\[
\begin{align*}
\text{CH} &\vdash \text{FTA}(PP') = \text{FT}_{PP'}^1 \\
\Omega &\vdash PP : I_1, PCT \xrightarrow{\text{certify}} PP', \text{FT}_{PP'}^1, PCT'
\end{align*}
\]

(6.2.9b) \( \Omega \vdash PP', I_2 \cdot CS, PPS', PCE' \xrightarrow{\text{certify}} PPS'', PCE'' \)

\[
\begin{align*}
\Omega &\vdash PP, I_1 \cdot I_2 \cdot CS, PPS, PCE \xrightarrow{\text{certify}} PPS'', PCE''
\end{align*}
\]

(6.2.9b.i) \( \Omega = \langle \Gamma, \_ , \text{FTA} \rangle \)

(6.2.9b.ii) \( \Gamma = \langle \_ , \text{CH} \rangle \)

(6.2.9b.iii) \( CE = \langle LS, FTC \rangle \)

(6.2.9b.iv) \( CE' = \langle LS', FTC \rangle \)

(6.2.9b.v) \( PCE = \langle P, CE \rangle \)

(6.2.9b.vi) \( PCE' = \langle P'', CE' \rangle \)

(6.2.9b.vii) \( PCT = \langle P', LS \rangle \)

(6.2.9b.viii) \( PCT' = \langle P'', LS' \rangle \)

(6.2.9b.ix) \( \text{FT}_{PP'}^1 = \text{FT}_{PP'}^1 \cap P(PP') \)

(6.2.9b.x) \( P' = P \cup \{ PP \rightarrow T \} \)

(6.2.9b.xi) \( \text{same side conditions as in (4.2.8b.i) through (4.2.8b.iii)} \).
\begin{align*}
\text{CH} & \vdash \text{FTA}(\text{PP}') \sqsubseteq \text{FT}^1_{\text{PP}'} \\
\Omega & \vdash \text{PP} : \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}^\text{PP} \cap \text{PCT} \\
\Omega & \vdash \text{PP}', \text{I}_2 \cdot \text{CS}, \text{PP'}, \text{PCE'} \xrightarrow{\text{certify}} \text{PP''}, \text{PCE''} \\
\Omega & \vdash \text{PP}, \text{I}_1 \cdot \text{I}_2 \cdot \text{CS}, \text{PPS}, \text{PCE} \xrightarrow{\text{certify}} \text{PP''}, \text{PCE''}
\end{align*}

(6.2.9c)

\begin{align*}
(6.2.9c.i) & \quad \text{where} \quad \Omega = \langle \Gamma, \omega, \text{FTA} \rangle \\
(6.2.9c.ii) & \quad \Gamma = \langle \omega, \text{CH} \rangle \\
(6.2.9c.iii) & \quad \text{CE} = \langle \text{LS}, \text{FTC} \rangle \\
(6.2.9c.iv) & \quad \text{CE'} = \langle \text{LS'}, \text{FTC'} \rangle \\
(6.2.9c.v) & \quad \text{PCE} = \langle \text{P}, \text{CE} \rangle \\
(6.2.9c.vi) & \quad \text{PCE'} = \langle \text{P''}, \text{CE'} \rangle \\
(6.2.9c.vii) & \quad \text{PCT} = \langle \text{P'}, \text{LS} \rangle \\
(6.2.9c.viii) & \quad \text{PCT'} = \langle \text{P''}, \text{LS'} \rangle \\
(6.2.9c.ix) & \quad \text{FT}^1_{\text{PP}'} = \text{FT}^\text{PP} \cap \text{P} \{ \text{PP}' \} \\
(6.2.9c.x) & \quad \text{FTC'} = \text{FTC} \cap \{ \text{PP}' \mapsto \text{FTA}(\text{PP}') \} \\
(6.2.9c.xi) & \quad \text{P'} = \text{P} \cup \{ \text{PP} \mapsto \top \} \\
(6.2.9c.xii) & \quad \text{same side conditions as in } (4.2.8b.i) \text{ through } (4.2.8b.iii).
\end{align*}

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 \text{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 \text{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 \text{P} is “garbage collected” in \text{PP}' by the code-sequence Rule 6.2.9c, only after the type constraint CH \vdash \text{FTA}(\text{PP}') \sqsubseteq \text{FT}^\text{PP} \cap \text{P} \{ \text{PP}' \} has been established, but before certification of the instruction at \text{PP}' has been performed.

We formulate an assumption which is invariant in the meaning of \text{PPS}, \text{LS}, \text{P}, \text{S}, \text{PP} and \text{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 \text{PPS}, \text{LS}, \text{P}, \text{S}, \text{PP}, and \text{FTA} are given as specified in our verification formalizations.
1. \( \text{PPS and LS are consistent with PP if all members are smaller than PP.} \)

2. \( \text{P is consistent with PP and FTA if PP' \leq PP implies } \text{P(PP')} = \top \text{ and PP'} > PP \text{ implies } \text{FTA(PP')} \subseteq \text{P(PP')} \).

3. \( \text{S is consistent with PP and FTA if PP' < PP implies } \text{S(PP')} = \text{FTA(PP)} \text{ and PP'} \geq PP \text{ implies } \text{S(PP')} = \bot. \)

4. \( \text{FTC is consistent with PP and FTA if PP' \leq PP implies FTC(PP') = FTA(PP') and PP' > PP implies FTC(PP') = \top.} \)

5. \( \text{A composite sort element is consistent with PP and FTA if all of its constituents are consistent with PP and FTA.} \)

**Lemma 6.2.14 (Sequence Verification Equivalences).** Let \( \text{PPS} \) 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 \( \Gamma, \Delta, \text{FTA, PP, C, and} \text{PPS', with} \text{PPS'} \text{consistent with} \text{PP}: \)

- \( (\Gamma, \Delta, \text{FTA}) \vdash_{\text{bv}} \text{PP, C, PPS' tsafe } \Rightarrow \text{PPS, and} \)
- \( \text{Find } (P', CE') \text{ and PPS' consistent with PP and FTA and a unique CE such that} \)
  \[
  (6.2.14a) \quad (\Gamma, \Delta, \text{FTA}) \vdash_{\text{lbv}} \text{PP, C, PPS', } (P', CE') \Rightarrow \text{PPS, } (P_0, CE)
  \]

Conversely, the following are equivalent for given \( \Gamma, \Delta, \text{PP, C, and P' and PPS' consistent with PP:} \)

- \( \text{Find } S' \text{ consistent with PP and FTA and } S \text{ consistent with max(PPS) such that} \)
  \[
  (6.2.14b) \quad (\Omega, \Delta, CE) \vdash_{\text{lbv}} \text{PP, FT, C, PPS', } (P', S') \Rightarrow \text{PPS, } (P_0, S)
  \]
- \( \text{Find FTA and CE' consistent with PP and FTA such that} \)
  \[
  (6.2.14c) \quad (\Gamma, \Delta, \text{FTA}) \vdash_{\text{lbv}} \text{PP, C, PPS', } (P', CE') \Rightarrow \text{PPS, } (P_0, CE)
  \]

**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 than 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.

### 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.
6.3. INSTRUCTION CERTIFICATION

Definition 6.3.1 (The Instruction Certification Signature). An instruction certification judgment has the signature

\[
\text{CodeContext} \vdash \text{bc} \quad \text{PPoint} : \text{Ins}, \text{PreConstr} \xrightarrow{\text{certify}} \text{PPoint}, \text{FrameType}, \text{PreConstr}
\]

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 \(\Omega\), on a set of pre-delayed constraints \(\text{PCT}\), with the expected frame type \(\text{FT}_{\text{PP}}\), in the successor point \(PP'\), and the accumulated pre-delayed constraint set \(\text{PCT}'\).

The certification rule for stack instructions can be obtained straightforwardly 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.

\[(6.3.2a)\]

\[
\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}} , \text{PCT}
\]

\[(6.3.2a.i)\] same side conditions as in (4.3.6a.i) through (4.3.6a.viii).

The certification rule for local variable instructions can be obtained straightforwardly 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.

\[(6.3.4a)\]

\[
\Theta \vdash \text{PP}, \text{ES}, \text{ET}, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}'
\]

\[
\Omega \vdash \text{PP} : \text{I}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \text{FT}_{\text{PP}} , \text{PCT}'
\]

\[(6.3.4a.i)\] 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.

\[
\begin{align*}
&\text{CH} \vdash T \subseteq \tau \\
&\Theta \vdash PP, ES, EHS, \text{PCT}^\text{certify} \rightarrow \text{PCT}' \\
&\Omega \vdash PP : 1, \text{PCT}^\text{certify} \rightarrow PP', \text{FT}_{PP'}^PP', \text{PCT}'
\end{align*}
\]

(6.3.6a)

(6.3.6a.i) where \( \text{PCT} = \langle P, LS \rangle \)

(6.3.6a.ii) \( \text{PCT}' = \langle P', LS' \rangle \)

(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.

\[
\begin{align*}
&\text{CH} \vdash T_1 \subseteq \tau_1 \\
&\vdots \\
&\text{CH} \vdash T_j \subseteq \tau_j \\
&\Theta \vdash PP, ES, EHS, \text{PCT}^\text{certify} \rightarrow \text{PCT}' \\
&\Omega \vdash PP : 1, \text{PCT}^\text{certify} \rightarrow PP', \text{FT}_{PP'}^PP', \text{PCT}'
\end{align*}
\]

(6.3.7a)

(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.

\[
\begin{align*}
(6.3.8a) & \quad \text{CH} \vdash \text{FTA}(\text{PP}^{''}) \sqsubseteq \text{FT}^{\text{PP}^{''}}_{\text{PP}^{''}} \\
& \quad \Omega \vdash \text{PP} : 1, \text{PCT} \xrightarrow{\text{certify}} \text{PP}^{'}', \text{FT}^{\text{PP}^{''}}_{\text{PP}^{''}} \xrightarrow{} \text{PCT}' \\
(6.3.8a.i) & \quad \text{where } \text{PP}^{''} \leq \text{PP} \\
(6.3.8a.ii) & \quad \text{PCT} = \langle \text{P}, \text{LS} \rangle \\
(6.3.8a.iii) & \quad \text{PCT}' = \langle \text{P}, \text{LS}' \rangle \\
(6.3.8a.iv) & \quad \text{LS}' = \text{LS} \cup \{\text{PP}^{''}\} \\
(6.3.8a.v) & \quad \text{same side conditions as in } (4.3.20a.i) \text{ through } (4.3.20a.vii). \\
\end{align*}
\]

\[
\begin{align*}
(6.3.8b) & \quad \text{CH} \vdash \text{FTA}(\text{PP}^{''}) \sqsubseteq \text{FT}^{\text{PP}^{''}}_{\text{PP}^{''}} \\
& \quad \Omega \vdash \text{PP} : 1, \text{PCT} \xrightarrow{\text{certify}} \text{PP}^{'}', \text{FT}^{\text{PP}^{''}}_{\text{PP}^{''}} \xrightarrow{} \text{PCT}' \\
(6.3.8b.i) & \quad \text{where } \text{PP}^{''} > \text{PP} \\
(6.3.8b.ii) & \quad \text{PCT} = \langle \text{P}, \text{LS} \rangle \\
(6.3.8b.iii) & \quad \text{PCT}' = \langle \text{P}', \text{LS} \rangle \\
(6.3.8b.iv) & \quad \text{P}' = \text{P} \setminus \{\text{PP}^{''} \rightarrow \text{FT}^{\text{PP}^{''}}\} \\
(6.3.8b.v) & \quad \text{same side conditions as in } (4.3.20a.i) \text{ through } (4.3.20a.vii). \\
\end{align*}
\]

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.

\[
\begin{align*}
(6.3.9a) & \quad \text{CH} \vdash \text{FTA}(\text{PP}^{''}) \sqsubseteq \text{FT}^{\text{PP}^{''}}_{\text{PP}^{''}} \\
& \quad \Omega \vdash \text{PP} : \text{goto}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}^{'}', \top, \text{PCT}' \\
(6.3.9a.i) & \quad \text{where } \text{PP}^{''} \leq \text{PP} \\
\end{align*}
\]
(6.3.9a.ii) \( \text{PCT} = \langle P, \text{LS} \rangle \)
(6.3.9a.iii) \( \text{PCT}' = \langle P, \text{LS}' \rangle \)
(6.3.9a.iv) \( \text{LS}' = \text{LS} \cup \{ \text{PP}'' \} \)
(6.3.9a.v) \( \text{same side-conditions as in (4.3.22a.i) through (4.3.22a.v)} \).

(6.3.9b) \[
\begin{align*}
\text{CH} \vdash & \text{FTA}(\text{PP}'') \subseteq \text{FT}^{\text{PP}''}_{\text{PP}''}, \\
\Omega \vdash & \text{PP} : \text{goto}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \top, \text{PCT}'
\end{align*}
\]

(6.3.9b.i) \( \text{where PP}'' > \text{PP} \)
(6.3.9b.ii) \( \text{PCT} = \langle P, \text{LS} \rangle \)
(6.3.9b.iii) \( \text{PCT}' = \langle P', \text{LS} \rangle \)
(6.3.9b.iv) \( P' = P \cap \{ \text{PP}'' \rightarrow \text{FT}^{\text{PP}''}_{\text{PP}''} \} \)
(6.3.9b.v) \( \text{same side-conditions as in (4.3.22a.i) through (4.3.22a.v)} \).

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) \[
\begin{align*}
\Theta \vdash & \text{PP}, \text{CID}_{E1}, \text{EHS}, \text{PCT} \rightarrow \text{PCT}' \\
\Omega \vdash & \text{PP} : \text{athrow}, \text{PCT} \xrightarrow{\text{certify}} \text{PP}', \top, \text{PCT}'
\end{align*}
\]

(6.3.10a.i) \( \text{same side conditions as in (4.3.24a.i) through (4.3.24a.viii)} \).

(6.3.10b) \( \text{same rule as in (4.3.24a)} \).

(6.3.10b.i) \( \text{same side conditions as in (4.3.24b.i) through (4.3.24b.viii)} \).

(6.3.10c) \( \text{same rule as in (4.3.24a)} \).

(6.3.10c.i) \( \text{same side conditions as in (4.3.24c.i) through (4.3.24c.vii)} \).

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.

\[
\Omega 
\vdash PP : return, PCT \xrightarrow{\text{certify}} PP', \top, PCT
\]

(6.3.11a)  
\begin{align*}
(6.3.11a.i) & \quad \text{same side-conditions as in (4.3.25a.i) through (4.3.25a.iii).} \\
(6.3.11b) & \quad \Omega \vdash PP : ireturn, PCT \xrightarrow{\text{certify}} PP', \top, PCT \\
(6.3.11b.i) & \quad \text{same side-conditions as in (4.3.25b.i) through (4.3.25b.iv).} \\
(6.3.11c) & \quad \Omega \vdash PP : areturn, PCT \xrightarrow{\text{certify}} PP', \top, PCT \\
(6.3.11c.i) & \quad \text{same side-conditions as in (4.3.25c.i) through (4.3.25c.v).}
\end{align*}

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.

\textbf{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.

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 method\(^2\).

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 \(FTA\):

\[
\Omega \vdash FTA(PP') \subseteq FT_{PP'} (BV\text{-invar})
\]

whereas the lightweight certification rules, however, impose the following constraint for the method’s program points:

\[
\Omega \vdash FTA(PP') \subseteq FT_{PP'} \cap P(PP') (LBC\text{-invar})
\]

By observation 6.3.12, we can assume that the frame types in the two constraints mean the same thing as they are given by the same notation.

\(\text{2A method code sequence contains at least one instruction.}\)
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Verifier rule</th>
<th>Certifier rules</th>
<th>Type constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>(4.3.6a)</td>
<td>(6.3.2a)</td>
<td>—</td>
</tr>
<tr>
<td>local variable</td>
<td>(4.3.8a)</td>
<td>(6.3.3a)</td>
<td>—</td>
</tr>
<tr>
<td>array</td>
<td>(4.3.11a)</td>
<td>(6.3.4a)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>simple</td>
<td>(4.3.13a)</td>
<td>(6.3.5a)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>field</td>
<td>(4.3.16a)</td>
<td>(6.3.6a)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>method</td>
<td>(4.3.17a)</td>
<td>(6.3.7a)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>branch</td>
<td>(4.3.20a)</td>
<td>(6.3.8a), (6.3.8b)</td>
<td>CH (\not\vdash) FTA(PP) (\subseteq) FT(PP)</td>
</tr>
<tr>
<td>goto</td>
<td>(4.3.22a)</td>
<td>(6.3.9a), (6.3.9b)</td>
<td>CH (\not\vdash) FTA(PP) (\subseteq) FT(PP)</td>
</tr>
<tr>
<td>athrow</td>
<td>(4.3.24a)</td>
<td>(6.3.10a)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>athrow</td>
<td>(4.3.24b)</td>
<td>(6.3.10b)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>athrow</td>
<td>(4.3.24c)</td>
<td>(6.3.10c)</td>
<td>Proposition 6.4.10</td>
</tr>
<tr>
<td>return</td>
<td>(4.3.25a)</td>
<td>(6.3.11a)</td>
<td>—</td>
</tr>
<tr>
<td>ireturn</td>
<td>(4.3.25b)</td>
<td>(6.3.11b)</td>
<td>—</td>
</tr>
<tr>
<td>areturn</td>
<td>(4.3.25c)</td>
<td>(6.3.11c)</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 6.2: Instruction rule type safety.

**Proof.** We reason over the set of program points by considering two separate cases.

1. \(PP' = 0\). The invariants are explicitly given as premises by the method rules in Definition 4.2.7 and Definition 6.2.4.

2. \(PP' > 0\). 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.

\(\square\)

**Observation 6.3.14 (P invariant).** We observe, that

- \(P(PP) = \top\) 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.

- \(P(PP) \neq \top\) 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, \(P(0) = \top\) 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.
6.3. INSTRUCTION CERTIFICATION

<table>
<thead>
<tr>
<th>Rule</th>
<th>Handling at PP''</th>
<th>Label update</th>
<th>Pending update</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6.3.2a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.3a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.4a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.5a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.6a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.7a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.8a)</td>
<td>backwards</td>
<td>LS ∪{PP''}</td>
<td>—</td>
</tr>
<tr>
<td>(6.3.8b)</td>
<td>forwards</td>
<td>—</td>
<td>P ∩{PP''} → FT_{pp''}</td>
</tr>
<tr>
<td>(6.3.9a)</td>
<td>backwards</td>
<td>LS ∪{PP''}</td>
<td>—</td>
</tr>
<tr>
<td>(6.3.9b)</td>
<td>forwards</td>
<td>—</td>
<td>P ∩{PP''} → FT_{pp''}</td>
</tr>
<tr>
<td>(6.3.10a)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.10b)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.10c)</td>
<td></td>
<td>Figure 6.7</td>
<td>Figure 6.7</td>
</tr>
<tr>
<td>(6.3.11a)</td>
<td></td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.3.11b)</td>
<td></td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.3.11c)</td>
<td></td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 6.3: Instruction certification updates.

**Case PP' = 0. (LBC-invar) ⇒ (BV-invar)** According to Observation 6.3.14, P(0) = T. Thus, by the lattice property of the frame type order, we have that CH ⊨ FTA(0) = FT₀ ∩ P(0) implies CH ⊨ FTA(0) ⊆ FT₀, and CH ⊨ FTA(0) ⊆ FT₀ ∩ P(0) implies CH ⊒ FTA(0) ⊆ FT₀.

**(BV-invar) ⇒ (LBC-invar)** By the lattice property of the frame type order, we can make the trivial extension FT_{pp'} = FT_{pp} ∩ T, Thus, by Observation 6.3.14, we immediately have CH ⊨ FTA(0) ⊆ FT₀ ∩ P(0).

**Case PP' > 0. (LBC-invar) ⇒ (BV-invar)** Follows immediately from Lemma 6.3.19.

**(BV-invar) ⇒ (LBC-invar)** According to Lemma 6.3.22 we have

(6.3.16) \( CH \vdash FTA(pp') \subseteq FT_{pp'} \)

(6.3.17) \( CH \vdash FTA(pp') \subseteq P(pp') \)

and consequently in (LBC) that

(6.3.18) \( CH \vdash FTA(pp') \subseteq FT_{pp'} \cap P(pp') \)

\[ \square \]

**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 ⊨ FTA(pp'') ⊆ P(pp'') is established in (LBC), CH ⊨ FTA(pp'') ⊆ FT_{pp''} is established in (BV).
Proof. When an (LBC)-rule establishes the constraint

\[(6.3.20) \quad CH \vdash FTA(PP'') \subseteq FT_{pp''} \cap P(PP')\]

for arbitrary program points PP, PP'', we have by the definition of \(\cap\) that

\[(6.3.21) \quad CH \vdash FTA(PP'') \subseteq FT_{pp''}\]

**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 \(CH \vdash FTA(PP'') \subseteq FT_{pp''}\) 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).

\[\square\]

**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 \(CH \vdash FTA(PP') \subseteq FT_{pp'}\) is established in (BV), then \(CH \vdash FTA(PP'') \subseteq P(PP'')\) 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 \(CH \vdash FTA(PP'') \subseteq FT_{pp''}\) 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 PP\((-1)\), PP\((-2)\), \ldots, PP\((-k)\). According to Figure 6.2, we have

\[(6.3.23) \quad CH \vdash FTA(PP'') \subseteq FT_{pp''}^{(-1)}\]

\[(6.3.24) \quad CH \vdash FTA(PP'') \subseteq FT_{pp''}^{(-2)}\]

\[(6.3.25) \quad \ldots\ldots\]

\[(6.3.26) \quad CH \vdash FTA(PP'') \subseteq FT_{pp''}^{(-k)}\]

By the lattice property of the frame type ordering follows that

\[(6.3.27) \quad CH \vdash FTA(PP'') \subseteq \bigcap_{j=1}^{k} FT_{pp''}^{(-j)}\]
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

\[
(6.3.28) \quad \prod_{j=1}^{k} F_{PPPP''}^{(P_{PPPP''})_{PPPP(j)}} = P(PP''')
\]

and consequently that

\[
(6.3.29) \quad CH \vdash FTA(PP''') \sqsubseteq P(PP''')
\]

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.

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).

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.
## CHAPTER 6. LIGHTWEIGHT CERTIFICATION FORMALIZATION

<table>
<thead>
<tr>
<th>Rule</th>
<th>Handling at PP''</th>
<th>Saved constraint</th>
<th>Pending update</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5.3.2a)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(5.3.3a)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(5.3.4a)</td>
<td>-</td>
<td>Figure 6.8</td>
<td>Figure 6.8</td>
</tr>
<tr>
<td>(5.3.5a)</td>
<td>-</td>
<td>Figure 6.8</td>
<td>Figure 6.8</td>
</tr>
<tr>
<td>(5.3.6a)</td>
<td>-</td>
<td>Figure 6.8</td>
<td>Figure 6.8</td>
</tr>
<tr>
<td>(5.3.7a)</td>
<td>-</td>
<td>Figure 6.8</td>
<td>Figure 6.8</td>
</tr>
<tr>
<td>(5.3.8a)</td>
<td>backwards</td>
<td>CH ⊢ S(PP'') ⊆ FT_{PP''}</td>
<td>-</td>
</tr>
<tr>
<td>(5.3.8b)</td>
<td>forwards</td>
<td>-</td>
<td>P ∩ {PP'' ↦ FF_{PP''}}</td>
</tr>
<tr>
<td>(5.3.9a)</td>
<td>backwards</td>
<td>CH ⊢ S(PP'') ⊆ FT_{PP''}</td>
<td>-</td>
</tr>
<tr>
<td>(5.3.10a)</td>
<td>-</td>
<td>Figure 6.8</td>
<td>Figure 6.8</td>
</tr>
<tr>
<td>(5.3.10b)</td>
<td>-</td>
<td>Figure 6.8</td>
<td>Figure 6.8</td>
</tr>
<tr>
<td>(5.3.11a)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(5.3.11b)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(5.3.11c)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**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 T-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 sequence 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

\[(6.3.33a)\]

\[
CH \vdash S(PP'') \subseteq FT_{PP''}
\]

and the rules which update the backward label set in (LBC), that is

\[(6.3.33b)\]

\[
LS \cup \{PP''\}
\]

**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.

\[\text{P is “garbage collected in PP by application of } P \sqcup \{PP \mapsto T\} \text{.}\]
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.

<table>
<thead>
<tr>
<th>Instruction group</th>
<th>stack</th>
<th>locals</th>
<th>array</th>
<th>simple</th>
<th>field</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certifying rule</td>
<td>(6.3.2a)</td>
<td>(6.3.3a)</td>
<td>(6.3.4a)</td>
<td>(6.3.5a)</td>
<td>(6.3.6a)</td>
<td>(6.3.7a)</td>
</tr>
<tr>
<td>Lightweight rule</td>
<td>(5.3.2a)</td>
<td>(5.3.3a)</td>
<td>(5.3.4a)</td>
<td>(5.3.5a)</td>
<td>(5.3.6a)</td>
<td>(5.3.7a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction group</th>
<th>branch</th>
<th>branch</th>
<th>goto</th>
<th>goto</th>
<th>throw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certifying rule</td>
<td>(6.3.8a)</td>
<td>(6.3.8b)</td>
<td>(6.3.9a)</td>
<td>(6.3.9b)</td>
<td>(6.3.10a)</td>
</tr>
<tr>
<td>Lightweight rule</td>
<td>(5.3.8a)</td>
<td>(5.3.8b)</td>
<td>(5.3.9a)</td>
<td>(5.3.9b)</td>
<td>(5.3.10a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction group</th>
<th>throw</th>
<th>throw</th>
<th>return</th>
<th>ireturn</th>
<th>areturn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certifying rule</td>
<td>(6.3.10b)</td>
<td>(6.3.10c)</td>
<td>(6.3.11a)</td>
<td>(6.3.11b)</td>
<td>(6.3.11c)</td>
</tr>
<tr>
<td>Lightweight rule</td>
<td>(5.3.10b)</td>
<td>(5.3.10c)</td>
<td>(5.3.11a)</td>
<td>(5.3.11b)</td>
<td>(5.3.11c)</td>
</tr>
</tbody>
</table>

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

$$CH \vdash S(PP'') \subseteq FT_{PP''}$$

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 $PP$).

\[\square\]
**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.

\[
\text{ExcContext} \vdash_{\text{bc}} \text{PPoint}, \text{Excs}, \text{Exc Handlers}, \text{Pre Constr} \overset{\text{certify}}{\rightarrow} \text{Pre Constr}
\]

where "\(\Theta \vdash \text{PP}, \text{EHS}, \text{EHS}, \text{PCT} \overset{\text{certify}}{\rightarrow} \text{PCT}'\)" reads: the set of compile-time exceptions, ES, which may be raised at the program point PP, verifies within the exception verification context \(\Theta\), 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.

\begin{align*}
(6.4.4a) & \quad \text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA} \\
& \quad \Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \text{PCT} \overset{\text{certify}}{\rightarrow} \text{PCT}
\end{align*}

(6.4.4a.i) *same side conditions as in (4.4.12a.i).*

\begin{align*}
(6.4.4b) & \quad \text{CH} \vdash_{\text{bv}} \text{PR}, \text{CID}_E, \text{EA} \\
& \quad \Theta \vdash \text{PP}, \text{ES}, \text{ET}, \text{PCT} \overset{\text{certify}}{\rightarrow} \text{PCT}'
\end{align*}

(6.4.4b.i) *same side conditions as in (4.4.12b.i) through (4.4.12b.ii).*

\begin{align*}
(6.4.4c) & \quad \Theta \vdash \text{PP}, \epsilon, \text{EHS}, \text{LS} \overset{\text{certify}}{\rightarrow} \text{LS}
\end{align*}

\begin{align*}
(6.4.4d) & \quad \Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EHS}, \text{PCT} \overset{\text{certify}}{\rightarrow} \text{PCT}' \\
& \quad \Theta \vdash \text{PP}, \text{CID}_E \cdot \text{ES}, \text{EH} \cdot \text{EHS}, \text{PCT} \overset{\text{certify}}{\rightarrow} \text{PCT}'
\end{align*}
same side conditions as in (4.4.12d.i) through (4.4.12d.ii).

\[
\Theta \vdash PP, CID_E \cdot ES, EHS, \text{PCT} \overset{\text{certify}}{\to} \text{PCT}'
\]

\[
\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS, \text{PCT} \overset{\text{certify}}{\to} \text{PCT}'
\]

(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.

\[
\text{CH} \vdash \text{FTA}(PP'') \subseteq \text{FT}_{PP''}^{PP}
\]

\[
\Theta \vdash PP, ES, ET, \text{PCT} \overset{\text{certify}}{\to} \text{PCT}'
\]

\[
\Theta \vdash PP, CID_E \cdot ES, EH \cdot EHS, \text{PCT} \overset{\text{certify}}{\to} \text{PCT}''
\]

(6.4.5a.i) where \( \text{PCT}' = \langle P', LS' \rangle \)

(6.4.5a.ii) \( \text{PCT}'' = \langle P'', LS'' \rangle \)

(6.4.5a.iii) \( LS'' = LS' \cup \{ PP'' \} \)

(6.4.5a.iv) \( PP'' \leq PP \)

(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) where \( \text{PCT}' = \langle P', LS' \rangle \)

(6.4.5b.ii) \( \text{PCT}'' = \langle P'', LS' \rangle \)

(6.4.5b.iii) \( PP'' > PP \)

(6.4.5b.iv) \( PP'' = P' \cap (PP'' \rightarrow \text{FT}_{PP}'') \)

(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.

\[
\begin{align*}
\mathbf{CH} & \vdash \text{FTA}(PP'') \sqsubseteq F_T^{PP''} \\
\Theta & \vdash PP, CID_E \cdot ES, EHS, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}' \\
\Theta & \vdash PP, CID_E \cdot ES, EH \cdot EHS, \text{PCT} \xrightarrow{\text{certify}} \text{PCT}''
\end{align*}
\]

(6.4.6a) \hspace{1cm} \text{where} \hspace{1cm} \text{PCT}' = \langle P', LS \rangle \\
(6.4.6a.i) \hspace{1cm} \text{PCT}'' = \langle P'', LS'' \rangle \\
(6.4.6a.ii) \hspace{1cm} LS'' = LS' \cup \{PP''\} \\
(6.4.6a.iii) \hspace{1cm} PP'' \leq PP \\
(6.4.6a.iv) \hspace{1cm} \text{same side conditions as in (4.4.16a.i) through (4.4.16a.vi).}

(6.4.6b) \hspace{1cm} \text{same rule as in (6.4.6a).}

(6.4.6b.i) \hspace{1cm} \text{where} \hspace{1cm} \text{PCT}' = \langle P', LS \rangle \\
(6.4.6b.ii) \hspace{1cm} \text{PCT}'' = \langle P'', LS'' \rangle \\
(6.4.6b.iii) \hspace{1cm} PP'' > PP \\
(6.4.6b.iv) \hspace{1cm} P'' = P' \cap \{PP'' \mapsto F_T^{PP''}\} \\
(6.4.6b.v) \hspace{1cm} \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 systems. 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.
### 6.4. EXCEPTION CERTIFICATION

<table>
<thead>
<tr>
<th>Verifying rule</th>
<th>Certifying rule</th>
<th>Catch</th>
<th>Type constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4.4.12a)</td>
<td>(6.4.4a)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12b)</td>
<td>(6.4.4b)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12c)</td>
<td>(6.4.4c)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12d)</td>
<td>(6.4.4d)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.12e)</td>
<td>(6.4.4e)</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>(4.4.14a)</td>
<td>(6.4.5a), (6.4.5b)</td>
<td>definite</td>
<td>CH ⊨ FTA(PP') ⊆ FT_{pp''}^{pp'}</td>
</tr>
<tr>
<td>(4.4.16a)</td>
<td>(6.4.6a), (6.4.6b)</td>
<td>potential</td>
<td>CH ⊨ FTA(PP') ⊆ FT_{pp''}^{pp'}</td>
</tr>
</tbody>
</table>

Figure 6.6: Exception rule type safety.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Handling at PP''</th>
<th>Label update</th>
<th>Pending update</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6.4.4a)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.4b)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.4c)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.4d)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.4e)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.5a)</td>
<td>backwards</td>
<td>LS ∪ (PP')</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.5b)</td>
<td>forwards</td>
<td>—</td>
<td>P ∩ (PP'' → FT_{pp''})^{pp'}</td>
</tr>
<tr>
<td>(6.4.6a)</td>
<td>backwards</td>
<td>LS ∪ (PP')</td>
<td>—</td>
</tr>
<tr>
<td>(6.4.6b)</td>
<td>forwards</td>
<td>—</td>
<td>P ∩ (PP'' → FT_{pp''})^{pp'}</td>
</tr>
</tbody>
</table>

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.

**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).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Handling at $PP''$</th>
<th>Saved Constraint</th>
<th>Pending update</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5.4.5a)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.5b)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.5c)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.5d)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.5e)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.6a)</td>
<td>backwards</td>
<td>$CH \vdash S(PP'') \subseteq FT_{PP''}$</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.6b)</td>
<td>forwards</td>
<td>—</td>
<td>$P \cap {PP'' \mapsto FT_{PP''}}$</td>
</tr>
<tr>
<td>(5.4.7a)</td>
<td>backwards</td>
<td>$CH \vdash S(PP'') \subseteq FT_{PP''}$</td>
<td>—</td>
</tr>
<tr>
<td>(5.4.7b)</td>
<td>forwards</td>
<td>—</td>
<td>$P \cap {PP'' \mapsto FT_{PP''}}$</td>
</tr>
</tbody>
</table>

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 $CH \vdash S(PP'') \subseteq FT_{PP''}$, 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 $CH \vdash S(PP'') \subseteq FT_{PP''}$ 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).

### 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.
Figure 6.9: No-catch exception-rule correspondence.

<table>
<thead>
<tr>
<th>Catch situation</th>
<th>Certifying rule</th>
<th>Lightweight rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(6.4.5a)</td>
<td>(5.4.6a)</td>
</tr>
<tr>
<td></td>
<td>(6.4.5b)</td>
<td>(5.4.6b)</td>
</tr>
<tr>
<td></td>
<td>(6.4.6a)</td>
<td>(5.4.7a)</td>
</tr>
<tr>
<td></td>
<td>(6.4.6b)</td>
<td>(5.4.7b)</td>
</tr>
</tbody>
</table>

Figure 6.10: Catch exception-rule correspondence.

**Proposition 6.5.1 (cksum lightweight certificates).** The `cksum()` method $M^{ck}$ lightweight certifies in the context $\Gamma^{ck}$ (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: $\Gamma^{ck} \vdash_{\text{light}} M^{ck}, FTA^{ck}, CE^{ck}$.

**Proof.** Assume $\Gamma^{ck}$, $M^{ck}$, $FTA^{ck}$, and $CE^{ck}$ as in the Proposition. Then the lightweight rules of this chapter can be used to construct the proof

\[
A.3.1 \\
\frac{\Omega^{ck}_{\text{light}} \vdash \langle 0, FTA^{ck} \rangle, c^{ck}, 0, \langle p^{ck}_0, S^{ck} \rangle \text{ ltsafe } p^{ck}_0, S^{ck}, 0 \rangle}{\Gamma^{ck} \vdash M^{ck}, CE^{ck}}
\]

where $\Omega^{ck}_{\text{light}}$, $\Delta^{ck}$, $p^{ck}_0$, $S^{ck}$, and $S^{ck}_{20}$ from Proposition 5.5.1, and where $A.3.1$ denotes the full proof tree given in Appendix A.3. 

\[\Box\]
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).
7.2. THE CLASS FILE CONTEXT

<table>
<thead>
<tr>
<th>Sort</th>
<th>definition</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>(3.1.4b)</td>
<td>int</td>
</tr>
<tr>
<td>ClassIdent</td>
<td>(3.2.2a)</td>
<td>String (internalized)</td>
</tr>
<tr>
<td>Identifier</td>
<td>(3.2.2k)</td>
<td>String</td>
</tr>
<tr>
<td>MaxStack</td>
<td>(3.2.6e)</td>
<td>int</td>
</tr>
<tr>
<td>MaxLocals</td>
<td>(3.2.6f)</td>
<td>int</td>
</tr>
<tr>
<td>PPoint</td>
<td>(3.2.8d)</td>
<td>int</td>
</tr>
<tr>
<td>Type</td>
<td>(3.2.2f)</td>
<td>String (internalized)</td>
</tr>
<tr>
<td>ReturnType</td>
<td>(3.2.2f)</td>
<td>String (internalized)</td>
</tr>
</tbody>
</table>

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.

```java
interface StdContext {
    // Class constant pool (cp).
    ConstPool getConstPool();

    // Class name (cid).
    String getClassIdent();

    // Class hierarchy (ch).
    ClassHier getClassHier();
}
```

Source 7.2.3 (ConstPool.java). Defines interface for implementations of the ConstPool and Item sorts of Definition 3.2.2.

```java
interface ConstPool {
    // Length of constant pool (max(Dom(cp))).
    int getConstPoolLength();

    // Get class identifier at index.
    // Throws VerifError if item is not a class identifier.
    String getClassIdent(int index) throws VerifError;

    // Get field reference at index.
    // Throws VerifError if item is not a field reference.
    FieldRef getFieldRef(int index) throws VerifError;

    // Get method reference at index.
    // Throws VerifError if item is not a method reference.
    MethRef getMethRef(int index) throws VerifError;
}
```
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 get BYU (int index) throws VerifError;
25
26 // Item in string form (for diagnostics).
27 String get item (int index); // item as string
28 }

Source 7.2.4 (FieldRef.java). Defines interface for implementations of the FieldRef sort of (3.2.2).

interface FieldRef {

    // Class containing field.
    String getClassIdent();

    // Field name.
    String getIdent();

    // Field type.
    String getType();

}

Source 7.2.5 (MethRef.java). Defines interface for implementations of the MethRef sort of (3.2.2c).

interface MethRef extends MethSig {

    // Class containing method.
    String getClassIdent();

    // Method name.
    String getIdent();

    // Method return type.
    String getReturnType();

}

Source 7.2.6 (MethSig.java). Defines interface for implementations of the MethSig sort of (3.2.2d).

// Implements
interface MethSig {

    // Name of method (id).
    String getIdent();

}
7.2. THE CLASS FILE CONTEXT

// Number of arguments (k).
int getArgumentCount();

// Argument type (t_i).
String getArgumentType(int i);

Source 7.2.7 (Method.java). Defines interface for implementations of the Method sort of (3.2.6a).
interface Method {
  // Method signature (msig).
  MethSig getMethSig();

  // Method return type (rt).
  String getReturnType();

  // Method code attribute (ca).
  CodeAtt getCodeAtt();

  // Method exception attribute (ea).
  ExcAtt getExcAtt();
}

Source 7.2.8 (ExcAtt.java). Defines interface for implementations of the ExcAtt sort of (3.2.6b).
interface ExcAtt {
  // Throws VerifError if excep is not in the exception attribute.
  void checkContains(String excep) throws VerifError;
}

Source 7.2.9 (CodeAtt.java). Defines interface for implementations of the CodeAtt and MaxFrame sorts of (3.2.6c) and 3.2.6d.
interface CodeAtt {
  // Size of stack [sic].
  int getMaxStack();

  // Number of local variables.
  int getMaxLocals();

  // Bytecode.
  Code getCode();

  // Table of exception handlers.
  ExcTable getExcTable();
}
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.

```java
interface Code {
    // Number of last byte in Code.
    int getMaxCode();

    // Opcode for bytecode instruction starting at pp.
    int getOpcode(int pp);

    // Length (in bytes) of bytecode instruction starting at pp.
    int getLength(int pp);

    // Return index or jump offset argument value of bytecode instruction starting at pp.
    int getArgument(int pp);

    // Printable representation of instruction starting at pp.
    String instruction(int pp);
}
```

Source 7.2.11 (ExcTable.java). Defines interface for implementations of the ExcTable and related sorts of Definition 3.2.12.

```java
interface ExcTable {
    // Number of exception handlers.
    int getExcCount();

    // Whether pp is within the range handled by handler number n.
    boolean inTryRange(int n, int pp);

    // Type handled by handler number n.
    String getCatchType(int n) throws VerifError;

    // Start of handler number n.
    int getCatchPP(int n);
}
```

Source 7.2.12 (ClassHier.java). Defines interface for implementations of the ClassHier sort and related ordering relation $\leq_{CH}$ of Section 3.3.

```java
interface ClassHier {
    // Whether a type is a (non-array) class in the class hierarchy.
    boolean isClass(String type);
}
```
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 $\subseteq$ following Definition 4.1.10.

```java
class Type {
    // Constant class names.
    final static String object = "Ljava/lang/Object;";
    final static String objectArray = "[Ljava/lang/Object;";
    final static String throwable = "Ljava/lang/Throwable;";
    final static String exception = "Ljava/lang/Exception;";
    final static String runtime = "Ljava/lang/RuntimeException;";
    final static String nullPointerException = "Ljava/lang/NullPointerException;";
    final static String arrayIndexOutOfBoundException = "Ljava/lang/ArrayIndexOutOfBoundsException;";
    final static String arrayStoreException = "Ljava/lang/ArrayStoreException;";
    final static String negativeArraySizeException = "Ljava/lang/NegativeArraySizeException;";
    final static String classCastException = "Ljava/lang/ClassCastException;";

    // Constant types.
    final static String integer = "I";
    final static String integerArray = "[I";

    // Abstract types.
    final static String nul = "null";
}
```
final static String top = "top";
final static String bot = "bot";
final static String botArray = "[bot";

// Void pseudo—types.
final static String voi = "V";

// Return type, i.e., type—internalized String, with name typeName.
// (All the following methods require (and return) such types.)
static String type(String typeName) {
    return typeName.intern();
}

// Whether type is the bot Type.
static boolean isBot(String type) {
    return (type == bot);
}

// Whether type is the null Type.
static boolean isNull(String type) {
    return (type == nul);
}

// Whether type is the void Type.
static boolean isVoid(String type) {
    return (type == voi);
}

// Whether type is an integer.
static boolean isInteger(String type) {
    return (type == integer);
}

// Whether type is the Object class.
static boolean isObject(String type) {
    return (type == object);
}

// Whether type is some array.
static boolean isArray(String type) {
    return (type != null && type.length() > 0 && type.charAt(0) == '[');
}

// Whether type is an array of integers.
static boolean isIntegerArray(String type) {
    return (type == integerArray);
}
7.3. LIGHTWEIGHT BYTECODE VERIFICATION

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 isClassArray(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       }
86  }
87  
88  // Whether type is the top Type.
89  static boolean isTop(String type) {
90       return (type == top);
91  }
92  
93  // Base Type of arrayType. Throws VerifError if not isArray(arrayType).
94  static String arrayBase(String arrayType) throws VerifError {
95       if (arrayType == botArray) return bot;
96       if (arrayType == objectArray) return object;
97       if (arrayType == integerArray) return integer;
98       return type(arrayType.substring(1)); // expensive hack: do better!
99  }
100  
101  // Array Type with elements of type.
102  static String arrayOf(String type) {
103       return type(”[" + type);
104  }
105  
106  // Whether t is a supertype of sub_t given the class hierarchy ch.
107  static boolean compatible(ClassHier ch, String t, String sub_t) {
108       try {
109            if (t == sub_t || t == bot || sub_t == top) return true;
110            if (ch.isClass(t) && ch.isClass(sub_t))
111                return ch.compatible(t, sub_t);
112            if (t == object && isArray(sub_t))
113                return true;
114            if (isArray(t) && isArray(sub_t))
115                return compatible(ch, arrayBase(t), arrayBase(sub_t));
116       }
117       catch (VerifError e) {} // cannot happen
118       return false;
// Largest type that is smaller than both t1 and t2.
static String meet(ClassHier ch, String t1, String t2) {
    String t = bot;
    try {
        if (t1 == t2) t = t1;
        else if (t1 == top) t = t2;
        else if (t2 == top) t = t1;
        else if (t1 == object && isArray(t2)) t = t1;
        else if (isArray(t1) && t2 == object) t = t2;
        else if (isArray(t1) && isArray(t2))
            t = arrayOf(meet(ch, arrayBase(t1), arrayBase(t2)));
        else if (ch.isClass(t1) && ch.isClass(t2))
            t = ch.meet(t1, t2);
    } catch (VerifError e) {} // cannot happen
    //System.out.println("meet(\"+t1+\",\"+t2+\") = \"+t");
    return t;
}

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.

class FrameType {

    // STATE.
    ClassHier ch;
    boolean isTop;
    boolean isBot;
    int stackLength;
    String[] stack;
    String[] locals;

    // CONSTRUCTORS.

    // Allocate uninitialized FrameType.
    FrameType(StdContext gamma, int ms, int ml) throws VerifError {
        ch = gamma.getClassHier();
    }
7.3. LIGHTWEIGHT BYTECODE VERIFICATION

```java
    stack = new String[ms];
    locals = new String[ml];
}

// Allocate FrameType with copy of another.
FrameType(FrameType ft) throws VerifError {
    ch = ft.ch;
    stack = new String[ft.stack.length];
    locals = new String[ft.locals.length];
    set(ft);
}

// INITIALIZATION METHODS.

// Set to the copy of another FrameType.
public void set(FrameType ft) {
    isTop = ft.isTop;
    isBot = ft.isBot;
    stackLength = ft.stackLength;
    for (int i = 0; i < stack.length; ++i) stack[i] = ft.stack[i];
    for (int i = 0; i < locals.length; ++i) locals[i] = ft.locals[i];
}

// Set to the top frame type.
public void setTop() {
    isTop = true;
    isBot = false;
    stackLength = 0;
    String bot = Type.bot;
    for (int i = 0; i < stack.length; ++i) stack[i] = bot;
    for (int i = 0; i < locals.length; ++i) locals[i] = bot;
}

// Set to the bottom FrameType.
public void setBot() {
    isTop = false;
    isBot = true;
    stackLength = 0;
    String bot = Type.bot;
    for (int i = 0; i < stack.length; ++i) stack[i] = bot;
    for (int i = 0; i < locals.length; ++i) locals[i] = bot;
}

// Set to the initial frame type corresponding to the class and method signature.
public void init(String cid, MethSig msig) throws VerifError {
    isTop = false;
    isBot = false;
```
stackLength = 0;

String bot = Type.bot;
for (int i = 0; i < stack.length; ++i) stack[i] = bot;

int argumentCount = msig.getArgumentCount();
if (argumentCount >= locals.length)
    throw VerifError.getInstance("cannot fit arguments in frame local variables");
locals[0] = cid;
for (int i = 1; i <= argumentCount; ++i)
    locals[i] = msig.getArgumentType(i - 1);
for (int i = argumentCount + 1; i < locals.length; ++i)
    locals[i] = bot;

// HIGH-LEVEL MUTATION METHODS.

// Update FrameType to be less than the other FrameType.
public void meetWith(FrameType other) {
    // System.out.println(" "+stringFrameType()+" meet "+other.stringFrameType()+" :");

    // Frame type unchanged if bottom or other is top.
    if (isBot || other.isTop) {
        // System.out.println(" = "+stringFrameType());
        return;
    }

    // FrameType forced to bottom if other is bottom or the stack/locals lengths differ.
    if (other.isBot || stack.length != other.stack.length || locals.length != other.locals.length)
        setBot();
        // System.out.println(" = "+stringFrameType());
        return;
    }

    // Frame type copied over if only this is top.
    if (isTop) {
        set(other);
        // System.out.println(" = "+stringFrameType());
        return;
    }

    // Otherwise build pointwise meet.
    for (int i = 0; i < stackLength; ++i)
        stack[i] = Type.meet(ch, stack[i], other.stack[i]);
    for (int i = 0; i < locals.length; ++i)
locals[i] = Type.meet(ch, locals[i], other.locals[i]);

//System.out.println(" = "+stringFrameType());

// BASIC ACCESS AND MUTATION METHODS.

// Mutate frame type to have one more type on the stack.
// Throws VerifError if this is impossible.
void stackPush(String type) throws VerifError {
  if (isBot || isTop)
    throw VerifError.getInstance("cannot push onto undefined or wrong stack");
  if (stackLength == stack.length)
    throw VerifError.getInstance("cannot push further onto full stack");
  stack[stackLength++] = type;
}

// Top stack type. Throws VerifError if stack is empty.
String stackTop() throws VerifError {
  if (isBot || isTop)
    throw VerifError.getInstance("undefined or wrong stack have no top element");
  if (stackLength == 0)
    throw VerifError.getInstance("no top element on empty stack");
  return stack[stackLength - 1];
}

// Mutate frame type to have one less type on the stack.
// Return the pop’d type.
// Throws VerifError if this is impossible.
String stackPop() throws VerifError {
  if (isBot || isTop)
    throw VerifError.getInstance("cannot pop from undefined or wrong stack");
  if (stackLength == 0)
    throw VerifError.getInstance("cannot pop from empty stack");
  String type = stack[--stackLength];
  stack[stackLength] = Type.bot;
  return type;
}

// Set stack to contain just a single type.
void stackSet(String type) throws VerifError {
  if (isBot || isTop)
    throw VerifError.getInstance("cannot set local variable of undefined or wrong frame");
  stack[0] = type;
  stackLength = 1;
}

// Set local variable n’s type. Throws VerifError if no such local variable.
void localsSet(int n, String type) throws VerifError {
    if (isBot || isTop)
        throw VerifError.getInstance("cannot set local variable of undefined or wrong frame");
    if (n < 0 || n >= locals.length)
        throw VerifError.getInstance("local variable number out of bounds");
    locals[n] = type;
}

// Local variable n's type. Throws VerifError if no such local variable.
String localsGet(int n) throws VerifError {
    if (isBot || isTop)
        throw VerifError.getInstance("cannot get local variable of undefined or wrong frame");
    if (n < 0 || n >= locals.length)
        throw VerifError.getInstance("local variable number out of bounds");
    return locals[n];
}

// CHECK FILTERS.
// Return first type argument.
// Throws VerifError if the first type is not the same as the second.
String checkSame(String type, String type2) throws VerifError {
    if (type != type2)
        throw VerifError.getInstance(type + " is not the same as " + type2);
    return type;
}

// Return first type argument.
// Throws VerifError if the first type is not a supertype of the second.
String checkSuper(String type, String subType) throws VerifError {
    if (!Type.compatible(ch, type, subType))
        throw VerifError.getInstance(type + " is not supertype of " + subType);
    return type;
}

// Return type argument.
// Throws VerifError if the first type is not a subtype of the second.
String checkSub(String type, String superType) throws VerifError {
    if (!Type.compatible(ch, superType, type))
        throw VerifError.getInstance(type + " is not subtype of " + superType);
    return type;
}

// String of frame type.
String stringFrameType() {
    StringBuffer sb = new StringBuffer();
    if (isBot)
7.3. LIGHTWEIGHT BYTECODE VERIFICATION

```java
sb.append("bot");
else if (isTop)
    sb.append("top");
else {
    sb.append("<");
    if (stackLength == 0)
        sb.append("*");
    else
        for (int i = 0; i < stackLength; ++i)
            sb.append((i == 0 ? "" : ",") + stack[i]);
    sb.append(">");
    if (locals.length == 0)
        sb.append("*");
    else
        for (int i = 0; i < locals.length; ++i)
            sb.append((i == 0 ? "" : ",") + locals[i]);
}
return sb.toString();
```

Source 7.3.4 (FrameTypeMap.java). Implements FrameTypeMap which provides the basis for the Pending and Saved sorts, cf. Definition 5.1.10.

```java
class FrameTypeMap {

    // State.

    private StdContext gamma;
    private int[] pps;
    private FrameType[] fts;

    // Constructor.

    // Construct FrameTypeMap with a maximal domain size.
    FrameTypeMap(StdContext gamma, int size) {
        this.gamma = gamma;
        pps = new int[size];
        for (int i = 0; i < size; ++i) pps[i] = -1;
        fts = new FrameType[size];
    }

    // Methods.

    // Set the mapping for pp to ft.
```
void set(int pp, FrameType ft) throws VerifError {
    for (int i = 0; i < pps.length; ++i)
        if (pps[i] == pp) {
            fts[i].meetWith(ft);
            return;
        }
    for (int i = 0; i < pps.length; ++i)
        if (pps[i] == -1) {
            pps[i] = pp;
            fts[i] = new FrameType(ft);
            fts[i].set(ft);
            return;
        }
    throw VerifError.getInstance("no more FrameType map slots");
}

boolean isEmpty() {
    for (int i = 0; i < pps.length; ++i)
        if (pps[i] != -1) return false;
    return true;
}

boolean contains(int pp) {
    for (int i = 0; i < pps.length; ++i)
        if (pps[i] == pp) return true;
    return false;
}

FrameType get(int pp) throws VerifError {
    for (int i = 0; i < pps.length; ++i)
        if (pps[i] == pp) return fts[i];
    throw VerifError.getInstance("map undefined for "+ pp);
}

void remove(int pp) throws VerifError {
    for (int i = 0; i < pps.length; ++i)
        if (pps[i] == pp) {
            pps[i] = -1;
            return;
        }
}
throw VerifError.getInstance("map undef for " + pp);
}

// Print frame type map.
String frameTypeMap() {
    if (pps.length == 0)
        return("{}");
    else {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < pps.length; ++i)
            if (pps[i] >= 0)
                sb.append((i==0 ? "": ",")
                    + pps[i] + ":"
                    + fts[i].stringFrameType()
                    + (i+1==pps.length?"":""));
        return sb.toString();
    }
}

Source 7.3.5 (Pending.java). Implements Saved sort (5.1.10c).

class Pending extends FrameTypeMap {
    // Construct Pending object for forward jumps cert.
    Pending(StdContext gamma, Cert cert, int pendingSize) {
        super(gamma, pendingSize);
    }
}

Source 7.3.6 (Saved.java). Implements Pending sort (5.1.10b).

class Saved extends FrameTypeMap {
    // Construct holder of saved frame types for backwards jumps labeled by cert.
    Saved(StdContext gamma, Cert cert) {
        super(gamma, cert.labelCount());
    }
}

Source 7.3.7 (LightContext.java). Interface to context of lightweight method verification of Definition 5.2.3 (implemented by 7.4.3 below).

class LightContext implements MethContext {
    // Fields.
    StdContext gamma;
String rt; // ReturnType
int ms; // MaxStack
int ml; // MaxLocals
ExcAtt ea;
ExcTable et;
Cert ce;

// Static instantiator...
private LightContext() {}
private static LightContext self = new LightContext();
static LightContext make(StdContext gamma, Method m, Cert ce) {
    self.gamma = gamma;
    self.rt = m.getReturnType();
    self.ms = ca.getMaxStack();
    self.ml = ca.getMaxLocals();
    self.ea = m.getExcAtt();
    self.et = ca.getExcTable();
    self.ce = ce;
    return self;
}

// Generic class context.
StdContext getStdContext() {
    return gamma;
}

// Method context.
MethContext getMethContext() {
    return this;
}

// Certificate.
Cert getCert() {
    return ce;
}

// MethContext implementation.
public String getReturnType() {
    return rt;
}

public int getMaxStack() {
    return ms;
}
7.3. LIGHTWEIGHT BYTECODE VERIFICATION

```java
public int getMaxLocals() {
    return ml;
}

public ExcAtt getExcAtt() {
    return ea;
}

public ExcTable getExcTable() {
    return et;
}
```

**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.

```java
interface Cert {
    // Number of labels?
    int labelCount();

    // Does the certificate label program point pp?
    boolean isLabel(int pp);

    // Modify ft to be equal to the smaller frame type that the certificate associates
    // to the pp program point. Throws VerifError if the modification would
    // not yield a smaller frame type.
    void applyDeltaTo(FrameType ft, int pp) throws VerifError;

    // Pending count.
    int getPendingSize();

    // Print certificate (for debugging).
    void printCert(String prefix);
}
```

**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.

```java
class MethLbv {
    // Verify method.
    static void verify(StdContext gamma, Method m, Cert ce, boolean verbose) throws VerifError {
        LightContext omega = LightContext.make(gamma, m, ce);
        MethContext delta = omega.getMethContext();
    }
```
CHAPTER 7. THE PROTOTYPE IMPLEMENTATION

FrameType ft = new FrameType(gamma, delta.getMaxStack(), delta.getMaxLocals());
ft.init(gamma.getClassIdent(), m.getMethSig());

// Access Code properties.
CodeAtt ca = m.getCodeAtt();
Code c = ca.getCode();
int pp_end = c.getMaxCode();

// Allocate saved and pending sets.
Saved S = new Saved(gamma, ce);
Pending P = new Pending(gamma, ce, ce.getPendingSize());

// Verify all instructions.
for (int pp = 0; pp <= pp_end; pp += c.getLength(pp)) {
    ce.applyDeltaTo(ft, pp);
    if (ce.isLabel(pp) || P.contains(pp)) {
        if (P.contains(pp)) {
            ft.meetWith(P.get(pp));
            P.remove(pp);
        }
        if (ce.isLabel(pp))
            S.set(pp, ft);
    }

    if (verbose) {
        System.out.println("-\w+", ft.stringFrameType());
        System.out.println(pp + "\w+", c.instruction(pp));
    }

    InsLbv.verify(omega, pp, ft, c, S, P);

    if (verbose)
        System.out.println("-\w+", ft.stringFrameType());

    if (!P.isEmpty())
        throw VerifError.getInstance("some pending constraints not resolved");
}

Source 7.3.10 (MethContext.java). Defines interface for MethContext sort of Definition 4.2.6.

Interface MethContext {

    // Method return type (rt).
String getReturnType();

// Method max stack length (ms).
int getMaxStack();

// Method max local variable length (ml).
int getMaxLocals();

// Method exception attribute (ea).
ExcAtt getExcAtt();

// Method exception table (et).
ExcTable getExcTable();

Source 7.3.11 (VerifError.java). Exception class used for lightweight bytecode verification failure.

class VerifError extends Exception {
    // Factory (permits singleton).
    static VerifError getInstance(String m) {
        return new VerifError(m);
    }
    
    // Constructor.
    private VerifError(String m) {
        super(m);
    }
}

Source 7.3.12 (InsLbv.java). Implements lightweight bytecode verification of a single instruction as specified in Section 5.3.

class InsLbv {
    // Verify instruction. Mutates ft, S, and P.
    // Throws VerifError on failure.
    static void verify(LightContext omega,
        int pp, FrameType ft, Code c, Saved S, Pending P)
        throws VerifError
    {
        StdContext gamma = omega.getStdContext();
        ClassHier ch = gamma.getClassHier();
        ConstPool cp = gamma.getConstPool();
        MethContext delta = omega.getMethContext();
    }
ExcTable et = delta.getExcTable();
ExcAtt ea = delta.getExcAtt();

Cert ce = omega.getCert();

switch (c.getOpcode(pp)) {
    // STACK INSTRUCTIONS.
    case 1: // aconst_null
        ft.stackPush(Type.nul);
        break;
    case 3: // iconst_0
    case 4: // iconst_1
        ft.stackPush(Type.integer);
        break;
    case 87: // pop
        ft.stackPop();
        break;
    case 89: // dup
        ft.stackPush(ft.stackTop());
        break;
    case 96: // iadd
    case 100: // isub
        ft.checkSame(ft.stackPop(), Type.integer);
        ft.checkSame(ft.stackTop(), Type.integer); // left as type of result
        break;
    // LOCAL VARIABLE INSTRUCTIONS.
    case 21: // iload
        ft.stackPush(ft.checkSame(ft.localsGet(c.getArgument(pp)), Type.integer));
        break;
    case 25: // aload
        ft.stackPush(ft.checkSub(ft.localsGet(c.getArgument(pp)), Type.object));
        break;
    case 54: // istore
        ft.localsSet(c.getArgument(pp), ft.checkSame(ft.stackPop(), Type.integer));
        break;
    case 58: // astore
7.3. LIGHTWEIGHT BYTECODE VERIFICATION

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\text{INT})
69 throw VerifError.getInstance("newarray\text{on}\text{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: // iload
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;
case 83: // aastore
    checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
    checkThrow(gamma, delta, ft, false, pp, Type.arrayIndexOutOfBoundsException, S, P);
    String valueType = ft.checkSub(ft.stackPop(), Type.object);
    ft.checkSame(ft.stackPop(), Type.integer);
    ft.checkSuper(Type.arrayBase(ft.stackPop()), valueType);
break;

// CONSTANT POOL INSTRUCTIONS.

case 18: // ldc (ldc_w)
    cp.getInteger(c.getArgument(pp)); // item must be integer within the constant pool
    ft.stackPush(Type.integer);
break;

case 180: // getfield
    checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
    FieldRef fref = cp.getFieldRef(c.getArgument(pp));
    ft.checkSub(ft.stackPop(), fref.getClassIdent());
    ft.stackPush(fref.getType());
break;

case 181: // putfield
    checkThrow(gamma, delta, ft, false, pp, Type.nullPointerException, S, P);
    FieldRef fref = cp.getFieldRef(c.getArgument(pp));
    ft.checkSub(ft.stackPop(), fref.getClassIdent());
    String rt = fref.getReturnType();
    if (!Type.isVoid(rt)) ft.stackPush(rt);
break;

case 182: // invokevirtual
    MethRef mref = cp.getMethRef(c.getArgument(pp));
    // Check arguments backwards
    for (int i = mref.getArgumentCount() - 1; i >= 0; --i)
        ft.checkSub(ft.stackPop(), mref.getArgumentType(i));
    ft.checkSub(ft.stackPop(), mref.getClassIdent());
    String rt = mref ReturnType();
    if (!Type.isVoid(rt)) ft.stackPush(rt);

    // Check all potential throws that might be caught by a handler...
    for (int i = 0; i < et.getExcCount(); ++i)
7.3. **LIGHTWEIGHT BYTECODE VERIFICATION**

```java
if (et.inTryRange(i, pp))
    checkThrow(gamma, delta, ft, false, pp, et.getCatchType(i), S, P);

break;

case 187: // new
    ft.stackPush(cp.getClassIdent(c.getArgument(pp)));
    break;

case 192: // checkcast
    checkThrow(gamma, delta, ft, false, pp, Type.classCastException, S, P);
    ft.stackPush(cp.getClassIdent(c.getArgument(pp)));
    ft.stackPush(Type.compatible(ch, toType, fromType) ? fromType : toType);
    break;

// JUMP INSTRUCTIONS.

case 153: // ifeq
    ft.checkSame(ft.stackPop(), Type.integer);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 154: // ifne
    ft.checkSub(ft.stackPop(), Type.object);
    ft.checkSub(ft.stackPop(), Type.object);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 155: // iflt
    ft.checkSame(ft.stackPop(), Type.integer);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 156: // ifgt
    ft.checkSame(ft.stackPop(), Type.integer);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 157: // ifge
    ft.checkSame(ft.stackPop(), Type.integer);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 158: // ificmpeq
    ft.checkSame(ft.stackPop(), Type.integer);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 159: // ifacmpeq
    ft.checkSub(ft.stackPop(), Type.object);
    ft.checkSub(ft.stackPop(), Type.object);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 160: // ificmep
    ft.checkSub(ft.stackPop(), Type.object);
    ft.checkSub(ft.stackPop(), Type.object);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 161: // ifacmpe
    ft.checkSub(ft.stackPop(), Type.object);
    ft.checkSub(ft.stackPop(), Type.object);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;
```

case 198: // ifnull
    ft.checkSub(ft.stackPop(), Type.object);
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    break;

case 167: // goto
    checkJump(gamma, delta, pp, pp+c.getArgument(pp), ft, S, P);
    ft.setTop();
    break;

// ABRUPT INSTRUCTIONS.

case 172: // ireturn
    ft.checkSame(ft.checkSame(ft.stackPop(), Type.integer), delta.getReturnType());
    ft.setTop();
    break;

case 176: // areturn
    ft.checkSub(ft.checkSub(ft.stackPop(), Type.object), delta.getReturnType());
    ft.setTop();
    break;

case 177: // return
    ft.setTop();
    break;

case 191: // athrow
    {
        String cid = ft.stackPop();
        //System.out.println(" athrow of "+cid);
        boolean pr = (ch.compatible(Type.exception, cid) & !ch.compatible(Type.runtime, cid));
        checkThrow(gamma, delta, ft, pr, pp, cid, S, P);
        ft.setTop();
    }
    break;

default:
    throw VerifError.getInstance("unknown instruction " + c.getOpcode(pp) + " at " + pp);
}

// Check single jump instruction in P and/or S.
static void checkJump(StdContext gamma, MethContext delta, int pp, int pp2,
                      FrameType ft, Saved S, Pending P) throws VerifError
7.3. LIGHTWEIGHT BYTECODE VERIFICATION

```java
{  
  if (pp2 > pp)  
    P.set(pp2, ft);  
  else  
    ft.meetWith(S.get(pp2));
}

// Check handling of exception e thrown at program point pp.
static void checkThrow(StdContext gamma, MethContext delta,  
  FrameType ft, boolean pr, int pp, String cid, e,  
  Saved S, Pending P) throws VerifError
{
  // Check local handlers first.
  ClassHier ch = gamma.getClassHier();  
  ExcTable et = delta.getExcTable();  
  for (int i = 0; i < et.getExcCount(); ++i) {
    if (et.inTryRange(i, pp)) {
      String cid = et.getCatchType(i);  
      boolean definite = (ch.compatible(cid, cid, e));  
      boolean potential = (ch.compatible(cid, e, cid));  
      if (definite || potential) {
        FrameType fte = new FrameType(ft);  
        fte.stackSet(cid, e);  
        checkJump(gamma, delta, pp, et.getCatchPP(i), fte, S, P);  
        if (definite) return;
      }
    }
  }

  // Check method exception table for checked exceptions.
  if (pr) {
    ExcAtt ea = delta.getExcAtt();  
    ea.checkContains(cid, e);
  }
}
```

**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_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.

```java
class LBV {

    // Lightweight bytecode verify class file named by argument.
    // Usage: "LBV [-v] class [method]"
    // "-v" option prints verbose messages to System.out.
    // "class" argument names class (without .class suffix).
    // "method" argument only method to verify.
    // Class hierarchy is read from "class.ch" and certificates from "class.method.cert".
    public static void main(String[] argv) {
        boolean failed = false;

        // Extract arguments or fail.
        boolean verbose = (argv.length >= 1 && argv[0].equals("-v"));
        int offset = (verbose ? 1 : 0);
        String className = (argv.length > offset ? argv[offset] : null);
        String onlyMethod = (argv.length > offset+1 ? argv[offset+1] : null);
        if (className == null || argv.length > offset+2) {
            System.err.println("Usage: java LBV [-v] class [method]);
            System.exit(2);
        }

        // Verification failure context.
        try {
            ClassInfo ci = BCEL.getClassInfo(className); // only system dependency!

            // Top—level context.
            StdContext gamma = ci.getStdContext();
            if (verbose) gamma.getClassHier().printClassHier("CLASS HIERARCHY ch=");

            // Verify methods one by one.
            for (int i = 0; i < ci.getMethodCount(); ++i) {
                ci.setMethodIndex(i);
                String method = ci.getMethodName();
                try {
                    if (onlyMethod == null || onlyMethod.equals(method)) {
                        if (verbose) System.out.println("METHOD " + ci.getMethodName() + "");
                    }
                } catch (Exception e) {
                    System.err.println("Method error: " + e.getMessage());
                }
            }

        } catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
        }
    }
}
```

176 CHAPTER 7. THE PROTOTYPE IMPLEMENTATION
7.4. THE INFRASTRUCTURE CODE

```java
Cert cert = ci.getCert();
if (verbose) cert.printCert("CERTIFICATE");

if (verbose) System.out.println("VERIFICATION");
Method m = ci.getMethod();
MethLbv.verify(gamma, m, cert, verbose);
if (verbose) System.out.println("VERIFICATION" + method + "; OK.");

catch (VerifError e) {
  if (verbose)
    System.out.println("VERIFICATION" + method + "; FAILED.");
  System.err.println(method + ";" + e.getMessage());
}

catch (VerifError e) {
  System.out.println("Initialization failed;" + e.getMessage() + ");
  failed = true;
}
if (failed) System.exit(1);
```

**Source 7.4.2 (ClassInfo.java).** Datastructure for main program.

```java
interface ClassInfo {

  // Verification context.
  StdContext getStdContext();

  // Number of methods in the class.
  int getMethodCount();

  // Set "current" method index.
  // Throws VerifError if the index is out of bounds 0..getMethodCount()–1.
  void setMethodIndex(int index) throws VerifError;

  // Get name of method with the set index.
  String getMethodName();

  // Get internal information of method with the set index.
  // Note: uses static storage so only use one at the time.
  Method getMethod();

  // Get certificate for method with the set index.
```
CHAPTER 7. THE PROTOTYPE IMPLEMENTATION

// Note: uses static storage so only use one at the time.
// Throws VerifError if there is no certificate for the method.
Cert getCert() throws VerifError;

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.

import java.io.*;
import java.util.Vector;
import java.util.BitSet;

class BCEL implements ClassInfo, StdContext, ConstPool, ClassHier,
Method, MethSig, CodeAtt, Code, ExcTable, ExcAtt, Cert,
de.fub.bytecode.Constants // include bytecode Constants
{
    // Singleton instance.
    private static BCEL self;

    // Creation factory method.
    public static ClassInfo getClassInfo(String name) throws VerifError {
        if (self == null) self = new BCEL();
        self.instantiate(name);
        return self;
    }

    // State.
    // ClassInfo state
    private int mCount; // number of methods
    private int mIndex; // "current" method index

    // StdContext state.
    private String file; // class file name
    private String cid; // class type
    private de.fub.bytecode.classfile.JavaClass jc; // BCEL class file object.
    private boolean certDirty; // whether certificate needs to be reread

    // ConstPool state.
    private de.fub.bytecode.classfile.ConstantPool jcp; // BCEL constant pool
    private int jcpLength; // BCEL constant pool length
    private FieldOrMethRef item; // cached BCEL constant pool item
// ClassHier state.
private String[] c; // class names
private int[] superIndex; // c[superIndex[i]] is the superclass of c[i], or bot.

// Method, MethSig, and CodeAtt state.
private de.fub.bytecode.classfile.Method jm; // cached BCEL method
private String desc; // cached method type description
private de.fub.bytecode.classfile.Code jcode; // cached BCEL code
private String[] jxs; // escaping exceptions
private de.fub.bytecode.classfile.CodeException[] jhs; // exception handlers

// Cert state.
private int[] certLabel;
private FrameTypeMap certFrameType;
private int certPendingSize;

// Private dummy constructor.
private BCEL() {}
skip("<:");

    supers.add(next_type());
    } while ((ch = next_nonspace()) ==");

    if (ch != "}") throw new IOException("", "} expected");

    catch (IOException x) {
        throw VerifError.getInstance("class hierarchy input error" + x.getMessage() + "}");
    }

    // Construct c array of class names.
    int clength = cs.size();
    c = new String[clength];
    for (int i = 0; i < clength; ++i) c[i] = (String) cs.elementAt(i);

    // Construct superIndex array of indices.
    int[] si = new int[clength];
    for (int i = 0; i < clength; ++i) {
        String supercsi = (String) supercs.elementAt(i);
        si[i] = findIndex(supercsi);
    }
    superIndex = si;

    // Read BCEL class object.
    try {
        //System.out.println( " Reading " + file + ".class file.");
        jc = new de.fub.bytecode.classfile.ClassParser(file + ".class").parse();
    }
    catch (IOException x) {
        throw VerifError.getInstance("could not read class file" + file + ".class" + x.getMessage() + "}");
    }

    // Get class name.
    cid = Type.type("L" + jc.getClassName() + ";");

    // Initialize constant pool.
    jcp = jc.getConstantPool();
    jcpLength = jcp.getLength();
    item = null;

    // Initialize method information.
    mCount = jc.getMethods().length;
    mIndex = -1;
    jm = null;
    desc = null;
    jcode = null;
7.4. THE INFRASTRUCTURE CODE

```java
jxs = null;
jhs = null;
certDirty = true;
}

// ClassInfo implementation.

public StdContext getStdContext() {
    return this;
}

public int getMethodCount() {
    return mCount;
}

public void setMethodIndex(int index) throws VerifError {
    if (index < 0 || index > getMethodCount())
        throw VerifError.getInstance("no method number " + index);
    mIndex = index;
    jm = jc.getMethods()[index];
    desc = jm.getSignature();
    jcode = jm.getCode();
    de.fub.bytecode.classfile.ExceptionTable jet = jm.getExceptionTable();
    if (jet == null)
        jxs = new String[0];
    else {
        jxs = new String[jet.getNumberOfExceptions()];
        int[] jxis = jet.getExceptionIndexTable();
        for (int i = 0; i < jxs.length; ++i)
            jxs[i] = Type.type(getClassIdent(jxis[i]));
        jhs = jcode.getExceptionTable();
        certDirty = true;
    }
}

public String getMethodName() {
    return jm.getName();
}

public Method getMethod() {
    return this;
}

public Cert getCert() throws VerifError {
    if (certDirty) {
        // Open certificate file for method.
    }
```
CHAPTER 7. THE PROTOTYPE IMPLEMENTATION

182
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228

try {
set input(file + ”.” + getMethodName() + ”.cert”);
}
catch (IOException x) {
// Fall−back: empty certificate.
certFrameType = new FrameTypeMap(this, 0);
certLabel = new int[0];
int certPendingSize = 3;
return this;
}
// Read certificate file for method.
try {
char ch;
skip(”(”);
// Read label list.
BitSet labels = new BitSet();
skip(”{”);
ch = next nonspace();
if (ch != ’}’) {
unread1(ch);
do {
//System.out.println(” Got label”);
labels.set(next integer());
} while ((ch = next nonspace()) == ’,’);
}
if (ch != ’}’) throw new IOException(”’,’ or ’}’ expected”);
int l = 0;
for (int i = 0; i < labels.length(); ++i) {
if (labels.get(i)) {
++l;
//System.out.println(” Got label #”+l+”: ”+i);
}
}
certLabel = new int[l];
for (int i = 0, j = 0; i < labels.length(); ++i)
if (labels.get(i)) {
//System.out.println(” Got label #”+i);
certLabel[j++] = i;
}
skip(”,”);
// Read FrameType ”deltas”.
Vector pps = new Vector();
Vector fts = new Vector();


```java
7.4. THE INFRASTRUCTURE CODE

skip("{");
ch = next_nonspace();
if (ch != '}') {
  unread1(ch);
  do {
    pps.add(new Integer(next_integer()));
    skip(":");
    FrameType ft = new FrameType(this, getStack(), getMaxLocals());
    skip("(");
    ch = next_nonspace();
    if (ch != ')') {f
      unread1(ch);
      do {
        ft.stackPush(next_type());
      } while ((ch = next_nonspace()) == ',');
    }
    if (ch != ')') throw new IOException("','	or
      ')' expected");
    skip(",");
  }
  skip("(");
  int i = 0;
  ch = next_nonspace();
  if (ch != ')') {f
    unread1(ch);
    do {
      ft.localsSet(i++, next_type());
    } while ((ch = next_nonspace()) == ',');
  }
  if (ch != ')') throw new IOException("','	or
    ')' expected");
  if (i != getMaxLocals())
    throw new IOException("incorrect local variable type numbers");
  fts.add(ft);
  skip(")");
  } while ((ch = next_nonspace()) == ',');
}
if (ch != '}') throw new IOException("','	or
  '}') expected");

certFrameType = new FrameTypeMap(this, pps.size());
for (int i = 0; i < pps.size(); ++i)
  certFrameType.set(((Integer) pps.get(i)).intValue(), (FrameType) fts.get(i));
skip(",");
certPendingSize = next_integer();
```
CHAPTER 7. THE PROTOTYPE IMPLEMENTATION

```java
276     skip("")
277     };
278     catch (IOException x) {
279         throw VerifError.getInstance("certificate input error" + x.getMessage() + ")");
280     }
281     certDirty = false;
282     return this;
283 }
284
285 // StdContext implementation.
286
287 public ConstPool getConstPool() {
288     return this;
289 }
290
291 public String getClassIdent() {
292     return cid;
293 }
294
295 public ClassHier getClassHier() {
296     return this;
297 }
298
299 // ConstPool implementation.
300
301 public int getConstPoolLength() {
302     return jcpLength;
303 }
304
305 public String getClassIdent(int index) throws VerifError {
306     if (index <= 0 || index >= jcpLength)
307         throw VerifError.getInstance("no constant pool item "+ index);
308     try {
309         de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
310         if (it.getTag() == CONSTANT_Class)
311             de.fub.bytecode.classfile.ConstantClass cc =
312                 (de.fub.bytecode.classfile.ConstantClass) it;
313             de.fub.bytecode.classfile.ConstantUtf8 cu8 =
314                 (de.fub.bytecode.classfile.ConstantUtf8) jcp.getConstant(cc.getNameIndex());
315             String t = cu8.getBytes();
316             return Type.type(t.charAt(0) == '[' ? t : "L" + t + "]");
317         } else
318             throw VerifError.getInstance("constant pool item number "+ index + ". is not ClassIdent");
319     }
320 ```
catch (java.lang.ClassFormatError x) {
    throw VerifError.getInstance("illegal item");
}

public FieldRef getFieldRef(int index) throws VerifError {
    try {
        de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
        if (it.getTag() == CONSTANT_Fieldref) {
            if (item == null) item = new FieldOrMethRef();
            item.setIndex(index);
            return (FieldRef) item;
        } else
            throw VerifError.getInstance("constant pool item number " + index + " is not FieldRef");
    }
    catch (java.lang.ClassFormatError x) {
        throw VerifError.getInstance("illegal item");
    }
}

public MethRef getMethRef(int index) throws VerifError {
    try {
        de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
        if (it != null && it.getTag() == CONSTANT_Methodref) {
            if (item == null) item = new FieldOrMethRef();
            item.setIndex(index);
            return (MethRef) item;
        } else
            throw VerifError.getInstance("constant pool item number " + index + " is not MethRef");
    }
    catch (java.lang.ClassFormatError x) {
        throw VerifError.getInstance("illegal item");
    }
}

public int getInteger(int index) throws VerifError {
    try {
        de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
        if (it.getTag() == CONSTANT_Integer)
            return ((de.fub.bytecode.classfile.ConstantInteger) it).getBytes();
        else
            throw VerifError.getInstance("constant pool item number " + index + " is not Integer");
    }
    catch (java.lang.ClassFormatError x) {
        throw VerifError.getInstance("no constant pool item number " + index);
public String getString(int index) throws VerifError {
    try {
        de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
        if (it.getTag() == CONSTANT_String) {
            int x = ((de.fub.bytecode.classfile.ConstantString) it).getStringIndex();
            de.fub.bytecode.classfile.ConstantUtf8 cu8 =
                (de.fub.bytecode.classfile.ConstantUtf8) jcp.getConstant(x);
            return cu8.getBytes();
        } else
            throw VerifError.getInstance("constant pool item number " + index + " is not String");
    }
    catch (java.lang.ClassFormatError x) {
        throw VerifError.getInstance("no constant pool item number " + index);
    }
}

public String getType(int index) throws VerifError {
    if (index < 0 || index >= jcpLength)
        throw VerifError.getInstance("no constant pool item #" + index);
    try {
        de.fub.bytecode.classfile.Constant it = jcp.getConstant(index);
        if (it.getTag() == CONSTANT_Utf8) {
            return Type.type("L" + ((de.fub.bytecode.classfile.ConstantUtf8) it).getBytes() + ";");
        } else if (it.getTag() == CONSTANT_Class) {
            return Type.type(getClassIdent(index));
        } else
            throw VerifError.getInstance("constant pool item number " + index + " is not Type");
    }
    catch (java.lang.ClassFormatError x) {
        throw VerifError.getInstance("illegal item " + x.getMessage() + ")");
    }
}

// String representation of ConstPool item.
public String item(int index) {
    StringBuffer it = new StringBuffer();
    try {
        de.fub.bytecode.classfile.Constant item = jcp.getConstant(index);
        if (item == null)
            return null;
    }
else if (item.getTag() == CONSTANT_Class)
    it.append(getClassIdent(index));
else if (item.getTag() == CONSTANT_Integer)
    it.append(getInteger(index));
else if (item.getTag() == CONSTANT_String)
    it.append("\"" + getString(index) + "\"");
else if (item.getTag() == CONSTANT_Fieldref) {
    FieldRef fref = getFieldRef(index);
    it.append("fieldref("+fref.getClassIdent()+","+fref.getIdent()+","+fref.getType()+")");
}
else if (item.getTag() == CONSTANT_Methodref) {
    MethRef mref = getMethRef(index);
    it.append("methodref("+mref.getClassIdent()+","+mref.getIdent()+","+mref.getReturnType()+")");
}
else
    return null;

try {
    VerifError x = null;
    it.append("?"+index);
    return it.toString();
} catch (VerifError x) {
    it.append("?"+index);
    return it.toString();
}

// FieldRef and MethRef implementation.
class FieldOrMethRef implements FieldRef, MethRef {

    // State.
    private de.fub.bytecode.classfile.ConstantCP it;
    private de.fub.bytecode.classfile.ConstantNameAndType nt;
    private String sig;
    void setIndex(int index) {
        it = (de.fub.bytecode.classfile.ConstantCP) jcp.getConstant(index);
        nt = (de.fub.bytecode.classfile.ConstantNameAndType)
            jcp.getConstant(it.getNameAndTypeIndex());
        sig = Type.type(((de.fub.bytecode.classfile.ConstantUtf8) jcp.getConstant(nt.getSignatureIndex())).getBytes());
    }

    public String getClassIdent() {
        int x = ((de.fub.bytecode.classfile.ConstantClass)
jcp.getConstant(it.getClassIndex()).getNameIndex();
return Type.type("L" + ((de.fub.bytecode.classfile.ConstantUtf8)
jcp.getConstant(x)).getBytes() + ":");

public String getIdent() {
    return Type.type(((de.fub.bytecode.classfile.ConstantUtf8)
jcp.getConstant(nt.getNameIndex())).getBytes());
}

public String getType() {
    return sig;
}

public String getReturnType() {
    return Type.type(de.fub.bytecode.generic.Type.getReturnType(sig).getSignature());
}

public int getArgumentCount() {
    return de.fub.bytecode.generic.Type.getArgumentTypes(sig).length;
}

public String getArgumentType(int i) {
    return Type.type(de.fub.bytecode.generic.Type.getArgumentTypes(sig)[i].getSignature());
}

// ClassHier implementation.

public boolean isClass(String type) {
    return (findIndex(type) >= 0);
}

int findIndex(String type) {
    for (int i = 0; i < c.length; ++i) {
        if (c[i] == type) {
            return i;
        }
    }
    return -1;
}

public boolean compatible(String cid, String sub_cid) throws VerifError {
    int i = findIndex(cid);
    if (i < 0)
        throw VerifError.getInstance("no class " + cid + " in class hierarchy");
    int sub_i = findIndex(sub_cid);
    if (sub_i < 0)
throw VerifError.getInstance("no class " + sub_i + " in class hierarchy");
while (sub_i != i && sub_i != -1)
    sub_i = superIndex[sub_i];
return (sub_i == i);
}

public String meet(String cid1, String cid2) throws VerifError {
    int i1 = findIndex(cid1);
    if (i1 < 0)
        throw VerifError.getInstance("no class " + cid1 + " in class hierarchy");
    int i2 = findIndex(cid2);
    if (i2 < 0)
        throw VerifError.getInstance("no class " + cid2 + " in class hierarchy");
    while (i1 > 0) {
        for (int i = i2; i > 0; i = superIndex[i])
            if (i1 == i) return c[i];
        i1 = superIndex[i1];
    }
    return Type.object;
}

// String representation of ClassHier.
public void printClassHier(String prefix) {
    for (int i = 0; i < c.length; ++i) {
        int si = superIndex[i];
        System.out.println((i == 0 ? prefix+"\" : ",") + c[i]+ ":\"");
        System.out.println((si < 0 ? "bot" : c[si]));
    }
    System.out.println("\"");
}

// Method implementation.
public MethSig getMethSig() {
    return this;
}

public String getReturnType() {
    return Type.type(de.fub.bytecode.generic.Type.getReturnType(desc).getSignature());
}

public CodeAtt getCodeAtt() {
    return this;
}

public ExcAtt getExcAtt() {
    return this;
}
// MethSig implementation.

public String getIdent() {
    return jm.getName();
}

public int getArgumentCount() {
    return de.fub.bytecode.generic.Type.getArgumentTypes(desc).length;
}

public String getArgumentType(int i) {
    return Type.type(de.fub.bytecode.generic.Type.getArgumentTypes(desc)[i].getSignature());
}

// CodeAtt implementation.

public int getMaxStack() {
    return jcode.getMaxStack();
}

public int getMaxLocals() {
    return jcode.getMaxLocals();
}

public Code getCode() {
    return this;
}

public ExcTable getExcTable() {
    return this;
}

// Code implementation.

public int getMaxCode() {
    return jcode.getCode().length - 1;
}

public int getOpcode(int pp) {
    int op = jcode.getCode()[pp];
    if (op < 0) op += 256;
    switch (op) {
        case WIDE:
            return getOpcode(pp+1);
7.4. THE INFRASTRUCTURE CODE

```java
public int getLength(int pp) {
    int op = jcode.getCode()[pp];
    if (op < 0) op += 256;
    switch (op) {
    case WIDE:
```
return 1 + getLength(pp+1);

case NEW:
    // Special case: new = NEW + DUP + INVOKEVIRTUAL <init> assumed!
    return 7;

case LDC_W:
case GETFIELD:
case PUTFIELD:
case INVOKEVIRTUAL:
case ANEWARRAY:
case CHECKCAST:
case IFEQ:
case IFNE:
case IFLT:
case IFGT:
case IFLE:
case IFGE:
case GOTO:
case IFNULL:
case IFNONNULL:
case SIPUSH:
    return 3;

case ILOAD:
case ALOAD:
case ISTORE:
case ASTORE:
case BIPUSH:
case NEWARRAY:
case LDC:
    return 2;

default:
    return 1;
}

public int getArgument(int pp) {
    int op = jcode.getCode()[pp];
    if (op < 0) op += 256;
    int b1, b2; // scratch variables
    switch (op) {
    case WIDE:
++pp; // hack: use 2-byte unsigned argument code below...

case LDC_W:

case GETFIELD:

case PUTFIELD:

case INVOKEVIRTUAL:

case NEW:

case ANEWARRAY:

case CHECKCAST:
    // Instructions with a two-byte unsigned argument.
    b1 = jcode.getCode()[pp+1];
    if (b1 < 0) b1 += 256;
    b2 = jcode.getCode()[pp+2];
    if (b2 < 0) b2 += 256;
    return (b1 + 256 + b2);

case IFEQ:

case IFNE:

case IFLT:

case IFGT:

case IFLE:

case IFGE:

case GOTO:

case IFNULL:

case IFNONNULL:
    // Instructions with a two-byte signed argument.
    b1 = jcode.getCode()[pp+1];
    b2 = jcode.getCode()[pp+2];
    if (b2 < 0) b2 += 256;
    return (b1 + 256 + b2);

case LDC:

case ILOAD:

case ALOAD:

case ISTORE:

case ASTORE:

case NEWARRAY:
    // Instructions with a single-byte unsigned argument.
    b1 = jcode.getCode()[pp+1];
    if (b1 < 0) b1 += 256;
    return b1;

    // Instructions with implied argument.

case ILOAD_0:

case ALOAD_0:

case ISTORE_0:

case ASTORE_0:
    return 0;
case ILOAD_1:
    break;
    
case ALOAD_2:
    return 2;
    
case ILOAD_3:
    case ALOAD_3:
    case ISTORE_3:
    case ASTORE_3:
        return 3;
    
    default:
        return 0;
    }
    }

// String representation of byte code instruction.
    public String instruction(int pp) {
        StringBuffer ins = new StringBuffer();
        int opcode = getOpcode(pp);
        int arg = getArgument(pp);

        switch (opcode) {
            case WIDE:
                switch (getOpcode(pp+1)) {
                case ILOAD: ins.append("wide iload "+arg); break;
                case ALOAD: ins.append("wide aload "+arg); break;
                case ISTORE: ins.append("wide istore "+arg); break;
                case ASTORE: ins.append("wide astore "+arg); break;
                }
                break;

            case ACONST_NULL: ins.append("aconst_null"); break;
            case ICONST_M1: ins.append("iconst -1"); break;
            case ICONST_0: ins.append("iconst 0"); break;
            case ICONST_1: ins.append("iconst 1"); break;
            case ICONST_2: ins.append("iconst 2"); break;

            default:
                return 0;
            }
        }
7.4. THE INFRASTRUCTURE CODE

case ICONST_3: ins.append("iconst_3"); break;
case ICONST_4: ins.append("iconst_4"); break;
case ICONST_5: ins.append("iconst_5"); break;
case POP: ins.append("pop"); break;
case DUP: ins.append("dup"); break;
case IADD: ins.append("iadd"); break;
case ISUB: ins.append("isub"); break;
case IRETURN: ins.append("ireturn"); break;
case ARETURN: ins.append("areturn"); break;
case RETURN: ins.append("return"); break;
case ATHROW: ins.append("athrow"); break;
case ARRAYLENGTH: ins.append("arraylength"); break;
case AASTORE: ins.append("aastore"); break;

806 case AALOAD: ins.append("aaload"); break;

807 case ILOAD: ins.append("iload_"+arg); break;
case ALOAD: ins.append("aload_"+arg); break;
case ISTORE: ins.append("istore_"+arg); break;
case ASTORE: ins.append("astore_"+arg); break;
case BIPUSH: ins.append("bipush_"+arg); break;
case SIPUSH: ins.append("sipush_"+arg); break;

811 case LDC: ins.append("ldc_"+item(arg)); break;
case LDC_W: ins.append("ldc_w_"+item(arg)); break;
case GETFIELD: ins.append("getfield_"+item(arg)); break;
case PUTFIELD: ins.append("putfield_"+item(arg)); break;
case INVOKEVIRTUAL: ins.append("invokevirtual_"+item(arg)); break;
case NEW: ins.append("new_"+item(arg)); break;
case NEWARRAY: ins.append("newarray_"+item(arg)); break;
case ANEWARRAY: ins.append("anewarray_"+item(arg)); break;
case CHECKCAST: ins.append("checkcast_"+item(arg)); break;

832 case IFEQ: ins.append("ifeq_"+(arg<0?""":"+")+arg); break;
case IFNE: ins.append("ifne_"+(arg<0?""":"+")+arg); break;
case IFLT: ins.append("iflt_"+(arg<0?""":"+")+arg); break;
case IFGT: ins.append("ifgt_"+(arg<0?""":"+")+arg); break;
case IFLE: ins.append("ifle_"+(arg<0?""":"+")+arg); break;
case IFGE: ins.append("ifge_"+(arg<0?""":"+")+arg); break;
case GOTO: ins.append("goto_"+(arg<0?""":"+")+arg); break;
case INULL: ins.append("ifnull_"+(arg<0?""":"+")+arg); break;
case IFNONNULL: ins.append("ifnonnull_"+(arg<0?""":"+")+arg); break;

default: ins.append("UNKNOWN_"+opcode+"_"+arg+")"); break;
}
return ins.toString();
}
public void checkContains(String excep) throws VerifError {
    for (int i = 0; i < jxs.length; ++i)
        if (compatible(jxs[i], excep))
            return;
    throw VerifError.getInstance("exception unhandled");
}

public int getExcCount() {
    return jhs.length;
}

public boolean inTryRange(int n, int pp) {
    return (jhs[n].getStartPC() <= pp && pp < jhs[n].getEndPC());
}

public String getCatchType(int n) throws VerifError {
    String t = Type.type(getClassIdent(jhs[n].getCatchType()));
    //System.out.println(" get	CatchType(\"+n++\") = "+t);
    return t;
}

public int getCatchPP(int n) {
    return jhs[n].getHandlerPC();
}

public boolean isLabel(int pp) {
    for (int i = 0; i < certLabel.length; ++i) {
        if (certLabel[i] == pp) return true;
    }
    return false;
}

public int labelCount() {
    return certLabel.length;
}

public void applyDeltaTo(FrameType ft, int pp) throws VerifError {
    if (certFrameType.contains(pp))
        ft.meetWith(certFrameType.get(pp));
}
public int getPendingSize() {
    return certPendingSize;
}

public void printCert(String prefix) {
    StringBuffer sb = new StringBuffer(prefix+"(");
    for (int i = 0; i < certLabel.length; ++i)
        sb.append((i==0?"":",") + certLabel[i]);
    sb.append(",");
    System.out.println(sb.toString());
    System.out.println(""+certFrameType.frameTypeMap()+",");
    System.out.println(""+certPendingSize+");
}

// Input utility methods.

// State.
private InputStreamReader input; // input stream
private int unread_char; // lookahead char

// Set input source.
private void setInput(String name) throws IOException {
    input = new FileReader(name);
    unread_char = -1;
}

// Return next character.
private char nextChar() throws IOException {
    char ch = read1();
    System.out.println(" read " + ch + ":");
    return ch;
}

// Return next non-space character.
private char nextNonspace() throws IOException {
    char ch = read1();
    while (Character.isWhitespace(ch)) ch = read1();
    System.out.println(" read " + ch + ":");
    return ch;
}

// Skip spaces and s.
private void skip(String s) throws IOException {
    unread1(read1Nonspace()); // hack to skip spaces before reading
    for (int i = 0; i < s.length(); ++i) {
        char ch = read1();
    }
if (ch != s.charAt(i)) throw new IOException(""" + s + "'expected""');

//System.out.println(" skip " + s + ":");

>Returns next integer on the input.

private int nextInteger() throws IOException {
    int i = 0;
    char ch;
    for (ch = read1nonspace(); Character.isDigit(ch); ch = read1()) {
        i = i*10 + Character.digit(ch, 10);
    }
    unread1(ch);
    //System.out.println(" integer " + i + ":");
    return i;
}

// Return next (extended) type identifier.

private String nextType() throws IOException {
    StringBuffer sb = new StringBuffer();
    char ch = read1nonspace();
    if (ch == '][') {
        sb.append(ch);
        ch = read1();
    } else if (ch == 'I') {
        sb.append('I');
    } else if (ch == 'b') {
        unread1(ch);
        skip("bot");
        sb.append("bot");
    } else if (ch == 't') {
        unread1(ch);
        skip("top");
        sb.append("bot");
    } else if (ch == 'L') {
        sb.append(ch);
        ch = read1();
    } while (Character.isJavaIdentifierStart(ch)) {
        do {
            sb.append(ch);
            while (Character.isJavaIdentifierPart(ch = read1())) sb.append(ch);
        } while (ch == '/');
    }
    if (ch != ":") throw new IOException(""";"'expected""');
7.4. THE INFRASTRUCTURE CODE

```java
sb.append(ch);
}
else throw new IOException("JVM_type_or_bot_or_top_expected");
return Type.type(sb.toString());
```

// Low-level character read.
```java
private char read1() throws IOException {
    if (unread_char >= 0) {
        char ch = (char) unread_char;
        unread_char = -1;
        return ch;
    } else {
        int ch = input.read();
        if (ch < 0)
            throw new IOException("end_of_file");
        else
            return (char) ch;
    }
}
```

// Low-level non-space character read.
```java
private char read1nonspace() throws IOException {
    char ch = read1();
    while (Character.isWhitespace(ch)) ch = read1();
    return ch;
}
```

// Low-level character unread.
```java
private void unread1(char ch) {
    unread_char = ch;
}
```

**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,
- \texttt{lbv.jar} file accompanying this thesis, and
- \texttt{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 \texttt{lbv.jar} and \texttt{bcel.jar} in the same directory, say $d$, and run

\begin{verbatim}
java -jar d/lbv.jar arguments
\end{verbatim}

where the possible arguments are given in the usage below.

Documentation 7.5.2 (Usage). The program takes the following arguments (in that sequence):

- \texttt{-v} Indicates that lbv should output the derived frame type as it goes. Optional.
- \texttt{class} The class to verify (without .class extension).
- \texttt{method} Method name to verify (by default all methods are verified). Optional.

In addition it expects to find these files:

- \texttt{class.ch} File describing class hierarchy for class context. Includes a \texttt{-delimited ,}-separated list of simple direct superclass relations using the JVM notation for class names and written “subclass $<$ superclass”.

- \texttt{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 \texttt{-delimited ,}-separated list of program point integers,
  - the frame type certificate as a \texttt{-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 \texttt{-separated list of extended type-name identifiers from the set I,bot,top,Lclass,; and \texttt{[type-name}, and
  - the largest size og 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 \texttt{cksum()} example is the following.

\begin{verbatim}
|
Ljava/lang/Throwable; $<$ Ljava/lang/Object; ,
Ljava/lang/Exception; $<$ Ljava/lang/Throwable; ,
Ljava/lang/RuntimeException; $<$ Ljava/lang/Exception; ,
Ljava/lang/NullPointerException; $<$ Ljava/lang/RuntimeException; ,
\end{verbatim}
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

\[
(8.1.1a) \quad CE = \{ (PP \mapsto FTA(PP)) \mid PP \in \{10, 15, 20, 39, 47\}, 0 \}
\]

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 \(P\)).
8.2. LEROY’S “ON-CARD VERIFIER”

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 $CE = \langle FTC, \varepsilon \rangle$ 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.

**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, $\exists LT : \forall PP, PP \geq PP_0 : FTA(PP) = \langle ST_{PP}, LT \rangle$.

2. Every jump target $PP$ satisfies $PP \geq PP_0$ and $FTA(PP) = \langle \varepsilon, LT \rangle$ (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 $PP_0 = 9$, $LT = \text{Gcd}11 \cdot \text{CrCardRd} \cdot \text{int} \cdot \text{int} \cdot \text{int}$, and the stack type is $\varepsilon$ 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 $\langle \varepsilon, LS \rangle$ such that lightweight verification succeeds.
```
<table>
<thead>
<tr>
<th>PP</th>
<th>FTA_{pp}.ST</th>
<th>FTA_{pp}.LT</th>
<th>(this \cdot ccnum \cdot x \cdot y \cdot z)</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot \perp \cdot \perp \cdot \perp</td>
<td>icont_0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot \perp \cdot \perp \cdot \perp</td>
<td>istore[2]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot \perp \cdot \perp</td>
<td>icont_0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot \perp \cdot \perp</td>
<td>istore[3]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot \perp</td>
<td>icont_0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot \perp</td>
<td>istore[4]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>aload[1]</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>CrCardRd</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>invokevirtual[1]</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[2]</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>goto[+8]</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>UnsetCrCard</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>pop</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>new[2]</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Abort</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>athrow</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>ldc_w[3]</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[3]</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[2]</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[3]</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>int \cdot int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>isub</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[4]</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[4]</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>ifle[+10]</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[4]</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[2]</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>goto[−16]</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[4]</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>ifne[+6]</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[+6]</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[2]</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[2]</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[4]</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[3]</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[2]</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>iload[4]</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>int</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>istore[3]</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>e</td>
<td>Gcd11 \cdot CrCardRd \cdot int \cdot int \cdot int</td>
<td>goto[−39]</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 8.1: checksum() that verifies with Leroy’s algorithm.
8.2. LEROY’S “ON-CARD VERIFIER”

Proof. The constraints clearly imply that a certificate can have the form \(\langle \text{LS}, \epsilon \rangle\), i.e., has an empty FTC component, as jumps are always between identical frame types \(\langle \epsilon, \text{LT} \rangle\). Thus the FTC component of any certificate will be empty. The certificate may contain an \text{LS} component but this corresponds to the fact that Leroy’s algorithm permits testing for whether a program point is a “jump target”.

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 \(\langle \epsilon, \text{LT} \rangle\). So one can optimize an implementation to not actually store these values but merely the program points in each of \(S\) and \(T\) 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.

Furthermore the Java virtual machine (JVM) specifies that field access control is performed through (dynamic) load and runtime 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.

```java
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

```java
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).

---

1. This 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).
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;
}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Accessibility from & Modifier & private & protected & public \\
\hline
same class & ✓ & ✓ & ✓ & ✓ \\
other class, same package & ✓ & ✓ & ✓ & ✓ \\
subclass outside package & ✓ & ✓ & ✓ & ✓ \\
other class outside package & ✓ & ✓ & ✓ & ✓ \\
\hline
\end{tabular}
\caption{Java Access Modifiers.}
\end{table}
(with c a class name, w and r field modifiers, t a field type, and f a field name) by

```java
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, §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].

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
</tr>
<tr>
<td>ACC_RD_PUBLIC</td>
<td>0x0100</td>
</tr>
<tr>
<td>ACC_RD_PRIVATE</td>
<td>0x0200</td>
</tr>
<tr>
<td>ACC_RD_PROTECTED</td>
<td>0x0400</td>
</tr>
</tbody>
</table>

Table 9.2: JVM read-only access_flags extension.

§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

```
private < package < protected < public
```
Assume a field is declared like

```java
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

```java
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

```
geteld(c1, w read r 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: \( w \) read \( r \) \( t \), 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 \models geteld(c1, w read r t, f)
```

defined by

```
c \models geteld(c1, w read public t, f)
```

\[ r \geq \text{protected} \quad c \leq: c_1 \]

```
c \models geteld(c1, w read r t, f)
```

\[ r \geq \text{package same-package}(c, c_1) \]

```
c \models geteld(c1, w read r t, f)
```

\[ c \models geteld(c, w read private t, f) \]

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

```java
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 \( \text{PP} \). 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 \( \langle e, LT \rangle \). 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 \( \langle e, LT \rangle \). 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 correponds 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 $c^k$-superscripts and impose a structure on the code code $c = iload[1] \cdot CS_0$ where $CS_0 = invokevirtual[1] \cdot CS_2, \ldots, CS_{57} = goto[-39]$ (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 $PP$ 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

$$\begin{array}{c}
\text{premise}_1 \quad \vdots \quad \text{premise}_k \\
\text{(Axiom)} \quad \vdots \quad \text{(Axiom)} \\
\text{conclusion} \quad \text{(Rule)}
\end{array}$$

for the inference proof fragment

$$\text{premise}_1 \quad \ldots \quad \text{premise}_k \quad \text{(Axiom)}$$

$$\text{conclusion} \quad \text{(Rule)}$$

A boxed premise is a reference to another definition where the proof of the premise(s) is given.

(A.1.2)

$$\begin{array}{c}
\Omega \vdash 0: iload[1] \rightarrow 2, FTA[2] \quad (4.3.8a) \\
CH \vdash FTA(2) \sqsubseteq FTA(2) \quad (4.1.28a) \\
\text{A.1.3} \\
\Omega \vdash 0, iload[1] \cdot invokevirtual[1] \cdot CS_2, \emptyset \Rightarrow PPS \quad (4.2.8b)
\end{array}$$

215
where $\Theta = (\Omega, \text{LT}_2, \text{false})$ with the contained list of possible exceptions obtained from $\{ E \mid E \leq_{\text{CH}} \text{Throwable} \}$, and with $\text{FTA}(10) = (\text{ST}_{10}, \text{LT}_{10})$.
where $\text{LT}_{11}$ is defined by $\text{FTA}(11) = \langle \omega, \text{LT}_{11} \rangle$.

where $\text{LT}_{14}$ is defined by $\text{FTA}(14) = \langle \omega, \text{LT}_{14} \rangle$.

where $\text{LT}_{15}$ is defined by $\text{FTA}(15) = \langle \omega, \text{LT}_{15} \rangle$. 
\[ \Omega \vdash 18: \text{istore}[3] \rightarrow 20, \text{FTA}(20) \tag{4.3.8a} \]
\[ \text{CH} \vdash \text{FTA}(20) \subseteq \text{FTA}(20) \tag{4.1.28a} \]
\[ \Omega \vdash 18, \text{istore}[3] \cdot \text{iload}[2] \cdot \text{CS}_{20}, [0, 2, \ldots, 15] \Rightarrow \text{PPS} \tag{4.2.8b} \]

\[ \Omega \vdash 20: \text{iload}[2] \rightarrow 22, \text{FTA}(22) \tag{4.3.8a} \]
\[ \text{CH} \vdash \text{FTA}(22) \subseteq \text{FTA}(22) \tag{4.1.28a} \]
\[ \Omega \vdash 20, \text{iload}[2] \cdot \text{iload}[3] \cdot \text{CS}_{22}, [0, 2, \ldots, 18] \Rightarrow \text{PPS} \tag{4.2.8b} \]

\[ \Omega \vdash 22: \text{iload}[3] \rightarrow 24, \text{FTA}(24) \tag{4.3.8a} \]
\[ \text{CH} \vdash \text{FTA}(24) \subseteq \text{FTA}(24) \tag{4.1.28a} \]
\[ \Omega \vdash 22, \text{iload}[3] \cdot \text{isub} \cdot \text{CS}_{24}, [0, 2, \ldots, 20] \Rightarrow \text{PPS} \tag{4.2.8b} \]

\[ \Omega \vdash 24: \text{isub} \rightarrow 25, \text{FTA}(25) \tag{4.3.6a} \]
\[ \text{CH} \vdash \text{FTA}(25) \subseteq \text{FTA}(25) \tag{4.1.28a} \]
\[ \Omega \vdash 24, \text{isub} \cdot \text{istore}[4] \cdot \text{CS}_{25}, [0, 2, \ldots, 22] \Rightarrow \text{PPS} \tag{4.2.8b} \]

\[ \Omega \vdash 25: \text{istore}[4] \rightarrow 27, \text{FTA}(27) \tag{4.3.8a} \]
\[ \text{CH} \vdash \text{FTA}(27) \subseteq \text{FTA}(27) \tag{4.1.28a} \]
\[ \Omega \vdash 25, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{27}, [0, 2, \ldots, 24] \Rightarrow \text{PPS} \tag{4.2.8b} \]

\[ \Omega \vdash 27: \text{iload}[3] \rightarrow 29, \text{FTA}(29) \tag{4.3.8a} \]
\[ \text{CH} \vdash \text{FTA}(29) \subseteq \text{FTA}(29) \tag{4.1.28a} \]
\[ \Omega \vdash 27, \text{iload}[3] \cdot \text{ifle}[+10] \cdot \text{CS}_{29}, [0, 2, \ldots, 25] \Rightarrow \text{PPS} \tag{4.2.8b} \]
A.1. STANDARD VERIFICATION PROOF

(A.1.16)
\[
\begin{align*}
& \text{CH} \vdash \text{FTA}(39) \subseteq \text{FTA}(32) \quad (4.1.28a) \\
& \Omega \vdash 29 : \text{ifle}[+10] \rightarrow 32, \text{FTA}(32) \quad (5.3.8b) \\
& \text{CH} \vdash \text{FTA}(32) \subseteq \text{FTA}(32) \quad (4.1.28a) \\
& \Omega \vdash 29, \text{ifle}[+10] : \text{iload}[4] \cdot \text{CS}_{32}, \{0, 2, \ldots, 27\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{align*}
\]

(A.1.17)
\[
\begin{align*}
& \Omega \vdash 32 : \text{iload}[4] \rightarrow 34, \text{FTA}(34) \quad (4.3.8a) \\
& \text{CH} \vdash \text{FTA}(34) \subseteq \text{FTA}(34) \quad (4.1.28a) \\
& \Omega \vdash 32, \text{iload}[4] : \text{istore}[2] \cdot \text{CS}_{34}, \{0, 2, \ldots, 29\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{align*}
\]

(A.1.18)
\[
\begin{align*}
& \Omega \vdash 34 : \text{istore}[2] \rightarrow 36, \text{FTA}(36) \quad (4.3.8a) \\
& \text{CH} \vdash \text{FTA}(36) \subseteq \text{FTA}(36) \quad (4.1.28a) \\
& \Omega \vdash 34, \text{istore}[2] : \text{goto}[+16] \cdot \text{CS}_{36}, \{0, 2, \ldots, 32\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{align*}
\]

(A.1.19)
\[
\begin{align*}
& \text{CH} \vdash \text{FTA}(20) \subseteq \text{FTA}(36) \quad (4.1.28a) \\
& \Omega \vdash 36 : \text{goto}[+16] \rightarrow 39, \uparrow \quad (4.3.22a) \\
& \text{CH} \vdash \text{FTA}(39) \subseteq \uparrow \quad (4.1.28d) \\
& \Omega \vdash 36, \text{goto}[+16] : \text{iload}[4] \cdot \text{CS}_{39}, \{0, 2, \ldots, 34\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{align*}
\]

(A.1.20)
\[
\begin{align*}
& \Omega \vdash 39 : \text{iload}[4] \rightarrow 41, \text{FTA}(41) \quad (4.3.8a) \\
& \text{CH} \vdash \text{FTA}(41) \subseteq \text{FTA}(41) \quad (4.1.28a) \\
& \Omega \vdash 39, \text{iload}[4] : \text{ifne}[+6] \cdot \text{CS}_{41}, \{0, 2, \ldots, 36\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{align*}
\]

(A.1.21)
\[
\begin{align*}
& \text{CH} \vdash \text{FTA}(47) \subseteq \text{FTA}(44) \quad (4.1.28a) \\
& \Omega \vdash 41 : \text{ifne}[+6] \rightarrow 44, \text{FTA}(44) \quad (4.3.20a) \\
& \text{CH} \vdash \text{FTA}(44) \subseteq \text{FTA}(44) \quad (4.1.28a) \\
& \Omega \vdash 41, \text{ifne}[+6] : \text{iload}[2] \cdot \text{CS}_{44}, \{0, 2, \ldots, 39\} \Rightarrow \text{PPS} \quad (4.2.8b)
\end{align*}
\]
\[\begin{align*}
\Omega \vdash 44 : \text{i1oad}[2] &\rightarrow 46, \text{FTA}(46) \tag{4.3.8a} \\
\text{CH} \vdash \text{FTA}(46) &\subseteq \text{FTA}(46) \tag{4.1.28a}
\end{align*}\]

(A.1.22)

\[\begin{align*}
\Omega \vdash 44, \text{i1oad}[2] &\cdot \text{ireturn} \cdot \text{CS}_{46}, [0, 2, \ldots, 41] \Rightarrow \text{PPS} \tag{4.2.8b}
\end{align*}\]

(A.1.23)

\[\begin{align*}
\Omega \vdash 46 : \text{ireturn} &\rightarrow 47, \top \tag{4.3.25} \\
\text{CH} \vdash \text{FTA}(47) &\subseteq \top \tag{4.1.28d}
\end{align*}\]

(A.1.24)

\[\begin{align*}
\Omega \vdash 46, \text{ireturn} &\cdot \text{i1oad}[2] \cdot \text{CS}_{47}, [0, 2, \ldots, 44] \Rightarrow \text{PPS} \tag{4.2.8b}
\end{align*}\]

(A.1.25)

\[\begin{align*}
\Omega \vdash 47 : \text{i1oad}[2] &\rightarrow 49, \text{FTA}(49) \tag{4.3.8a} \\
\text{CH} \vdash \text{FTA}(49) &\subseteq \text{FTA}(49) \tag{4.1.28a}
\end{align*}\]

(A.1.26)

\[\begin{align*}
\Omega \vdash 47, \text{i1oad}[2] \cdot \text{istore}[4] \cdot \text{CS}_{49}, [0, 2, \ldots, 46] \Rightarrow \text{PPS} \tag{4.2.8b}
\end{align*}\]

(A.1.27)

\[\begin{align*}
\Omega \vdash 49 : \text{istore}[4] &\rightarrow 51, \text{FTA}(51) \tag{4.3.8a} \\
\text{CH} \vdash \text{FTA}(51) &\subseteq \text{FTA}(51) \tag{4.1.28a}
\end{align*}\]

(A.1.28)

\[\begin{align*}
\Omega \vdash 49, \text{istore}[4] &\cdot \text{i1oad}[3] \cdot \text{CS}_{51}, [0, 2, \ldots, 47] \Rightarrow \text{PPS} \tag{4.2.8b}
\end{align*}\]

(A.1.29)

\[\begin{align*}
\Omega \vdash 51 : \text{i1oad}[3] &\rightarrow 53, \text{FTA}(53) \tag{4.3.8a} \\
\text{CH} \vdash \text{FTA}(53) &\subseteq \text{FTA}(53) \tag{4.1.28a}
\end{align*}\]

(A.1.30)

\[\begin{align*}
\Omega \vdash 51, \text{i1oad}[3] \cdot \text{istore}[2] \cdot \text{CS}_{53}, [0, 2, \ldots, 49] \Rightarrow \text{PPS} \tag{4.2.8b}
\end{align*}\]

(A.1.31)

\[\begin{align*}
\Omega \vdash 53 : \text{istore}[2] &\rightarrow 55, \text{FTA}(55) \tag{4.3.8a} \\
\text{CH} \vdash \text{FTA}(55) &\subseteq \text{FTA}(55) \tag{4.1.28a}
\end{align*}\]

(A.1.32)

\[\begin{align*}
\Omega \vdash 53, \text{istore}[2] &\cdot \text{i1oad}[4] \cdot \text{CS}_{55}, [0, 2, \ldots, 51] \Rightarrow \text{PPS} \tag{4.2.8b}
\end{align*}\]

(A.1.33)
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) \[
\Omega \vdash 55 : \text{iload}[4] \rightarrow 57, \text{FTA}(57) \quad (A.28)
\]

\[
\text{CH} \vdash \text{FTA}(57) \subseteq \text{FTA}(57) \quad (A.29)
\]

\[
\Omega \vdash 55, \text{iload}[4] \cdot \text{istore}[3] \cdot \text{CS}_{57}, \{0, 2, \ldots, 53\} \Rightarrow \text{PPS} \quad (4.2.8b)
\]

(A.2.2) \[
\Omega \vdash 57 : \text{istore}[3] \rightarrow 59, \text{FTA}(59) \quad (A.29)
\]

\[
\text{CH} \vdash \text{FTA}(59) \subseteq \text{FTA}(59) \quad (A.30)
\]

\[
\Omega \vdash 57, \text{istore}[3] \cdot \text{goto}[-39] \cdot \epsilon, \{0, 2, \ldots, 55\} \Rightarrow \text{PPS} \quad (4.2.8b)
\]

(A.30) \[
\Omega \vdash 59 : \text{goto}[-39] \rightarrow 62, \quad (4.22a)
\]

\[
\Omega \vdash 59, \text{goto}[-39], \{0, 2, \ldots, 55, 57\} \Rightarrow \text{PPS} \quad (4.2.8a)
\]
where $P_2 = P_0 \cap \{10 \mapsto \text{FTA}(10)\}$ here and below, and with $\Theta_{\text{light}} = \langle \Omega_{\text{light}}, \text{LT}_2, \text{false} \rangle$ with the contained list of possible exceptions obtained from $\{E | E \leq_{\text{CH}} \text{Throwable} \}$, and with $\text{FTA}(10) = \langle \text{ST}_{10}, \text{LT}_{10} \rangle$.

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 2, \text{FTA}(2) \rangle, \text{invokevirtual}[1] \cdot \text{istore}[2] \cdot \text{CS}_5, \langle 0 \rangle, \langle P_0, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \tag{5.2.6b}
\end{align*}

where $P_2 = P_0 \cap \{10 \mapsto \text{FTA}(10)\}$ here and below.

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 5, \text{FTA}(5) \rangle, \text{istore}[2], \langle P_2, S_0 \rangle \rightarrow \langle 7, \text{FTA}(7) \rangle, P_2 \tag{5.3.3a}
\end{align*}

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 5, \text{FTA}(5) \rangle, \text{istore}[2] \cdot \text{goto}[+8] \cdot \text{CS}_7, \langle 0, 2 \rangle, \langle P_2, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \tag{5.2.6b}
\end{align*}

where $P_7 = P_2 \cap \{15 \mapsto \text{FTA}(15)\}$ here and below.

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 7, \text{FTA}(7) \rangle, \text{goto}[+8], \langle P_2, S_0 \rangle \rightarrow \langle 10, \top \rangle, P_7 \tag{5.3.9b}
\end{align*}

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 7, \text{FTA}(7) \rangle, \text{goto}[+8] \cdot \text{pop} \cdot \text{CS}_1, \langle 0, 2, 5 \rangle, \langle P_2, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \tag{5.2.6b}
\end{align*}

where $P_7 = P_2 \cap \{15 \mapsto \text{FTA}(15)\}$ here and below.

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 10, \text{FTA}(10) \rangle, \text{pop}, \langle P_7, S_0 \rangle \rightarrow \langle 11, \text{FTA}(11) \rangle, P_{10} \tag{5.3.2a}
\end{align*}

\begin{align*}
\Omega_{\text{light}} &\vdash \langle 10, \top \rangle, \text{pop} \cdot \text{new}[2] \cdot \text{CS}_1, \langle 0, 2, 5, 7 \rangle, \langle P_{10}, S_0 \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \tag{5.2.6b}
\end{align*}
where $P_{10} = P_7 \cap \{10 \rightarrow T\}$.

\[(A.2.6)\]
\[
\begin{align*}
\Omega_{\text{light}} \vdash \langle 11, \text{FTA}(11) \rangle, \text{new}[2], \langle P_{10}, S_0 \rangle & \rightarrow \langle 14, \text{FTA}(14) \rangle, P_{10} \\
\Omega_{\text{light}} \vdash \langle 11, \text{FTA}(11) \rangle & \rightarrow \langle 14, \text{FTA}(14) \rangle, P_{10}
\end{align*}
\]

where $LT_{11}$ is defined by $\text{FTA}(11) = \langle \_ , LT_{11} \rangle$.

\[(A.2.7)\]
\[
\begin{align*}
\Omega_{\text{light}} \vdash \langle 11, \text{FTA}(11) \rangle, \text{new}[2] \cdot \text{athrow} \cdot \text{CS}_{14}, [0, \ldots, 10], \langle P_{10}, S_0 \rangle & \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \\
\Omega_{\text{light}} \vdash \langle 11, \text{FTA}(11) \rangle & \rightarrow \langle 14, \text{FTA}(14) \rangle, P_{10}
\end{align*}
\]

where $LT_{14}$ is defined by $\text{FTA}(14) = \langle \_ , LT_{14} \rangle$.

\[(A.2.8)\]
\[
\begin{align*}
\Omega_{\text{light}} \vdash \langle 14, \text{FTA}(14) \rangle, \text{athrow} \cdot \text{ldc} \cdot \text{w}[3] \cdot \text{CS}_{15}, [0, \ldots, 11], \langle P_{10}, S_0 \rangle & \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \\
\Omega_{\text{light}} \vdash \langle 14, \text{FTA}(14) \rangle & \rightarrow \langle 15, \top \rangle, P_{10}
\end{align*}
\]

where $LT_{15}$ is defined by $\text{FTA}(15) = \langle \_ , LT_{15} \rangle$.

\[(A.2.9)\]
\[
\begin{align*}
\Omega_{\text{light}} \vdash \langle 15, \text{FTA}(15) \rangle, \text{ldc} \cdot \text{w}[3] \cdot \text{istore} \cdot \text{CS}_{18}, [0, \ldots, 14], \langle P_0, S_0 \rangle & \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \\
\Omega_{\text{light}} \vdash \langle 15, \text{FTA}(15) \rangle & \rightarrow \langle 18, \text{FTA}(18) \rangle, P_0
\end{align*}
\]

where $LT_{15}$ is defined by $\text{FTA}(15) = \langle \_ , LT_{15} \rangle$.

\[(A.2.10)\]
\[
\begin{align*}
\Omega_{\text{light}} \vdash \langle 18, \text{FTA}(18) \rangle, \text{istore} \cdot \text{CS}_{20}, [0, \ldots, 15], \langle P_0, S_0 \rangle & \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \\
\Omega_{\text{light}} \vdash \langle 18, \text{FTA}(18) \rangle & \rightarrow \langle 20, \text{FTA}(20) \rangle, P_0
\end{align*}
\]
\[(A.2.10)\]
\[
\Omega_{\text{light}} \vdash \langle 20, \text{FTA}(20) \rangle, \text{i}0\text{ad}[2], \langle P_0, S_{20} \rangle \rightarrow \langle 22, \text{FTA}(22) \rangle, P_0 \quad (5.3.3a)
\]
\[
\Omega_{\text{light}} \vdash \langle 20, \text{FTA}(20) \rangle, \text{i}0\text{ad}[2] \cdot \text{i}0\text{oad}[3] \cdot \text{CS}_{22}, \{0, \ldots, 18\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \quad (5.2.6b)
\]

\[(A.2.11)\]
\[
\Omega_{\text{light}} \vdash \langle 22, \text{FTA}(22) \rangle, \text{i}0\text{oad}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 24, \text{FTA}(24) \rangle, P_0
\]
\[
\Omega_{\text{light}} \vdash \langle 22, \text{FTA}(22) \rangle, \text{i}0\text{oad}[3] \cdot \text{i}0\text{sub} \cdot \text{CS}_{24}, \{0, \ldots, 20\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \quad (5.2.6b)
\]

\[(A.2.12)\]
\[
\Omega_{\text{light}} \vdash \langle 24, \text{FTA}(24) \rangle, \text{i}0\text{sub}, \langle P_0, S_{20} \rangle \rightarrow \langle 25, \text{FTA}(25) \rangle, P_0
\]
\[
\Omega_{\text{light}} \vdash \langle 24, \text{FTA}(24) \rangle, \text{i}0\text{sub} \cdot \text{i}0\text{store}[4] \cdot \text{CS}_{25}, \{0, \ldots, 22\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \quad (5.2.6b)
\]

\[(A.2.13)\]
\[
\Omega_{\text{light}} \vdash \langle 25, \text{FTA}(25) \rangle, \text{i}0\text{store}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 27, \text{FTA}(27) \rangle, P_0
\]
\[
\Omega_{\text{light}} \vdash \langle 25, \text{FTA}(25) \rangle, \text{i}0\text{store}[4] \cdot \text{i}0\text{oad}[3] \cdot \text{CS}_{27}, \{0, \ldots, 24\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \quad (5.2.6b)
\]

\[(A.2.14)\]
\[
\Omega_{\text{light}} \vdash \langle 27, \text{FTA}(27) \rangle, \text{i}0\text{oad}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 29, \text{FTA}(29) \rangle, P_0
\]
\[
\Omega_{\text{light}} \vdash \langle 27, \text{FTA}(27) \rangle, \text{i}0\text{oad}[3] \cdot \text{i}0\text{fl}[+10] \cdot \text{CS}_{29}, \{0, \ldots, 25\}, \langle P_0, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_0, S_{20} \rangle \quad (5.2.6b)
\]

\[(A.2.15)\]
\[
\Omega_{\text{light}} \vdash \langle 29, \text{FTA}(29) \rangle, \text{i}0\text{fl}[+10], \langle P_0, S_{20} \rangle \rightarrow \langle 32, \text{FTA}(32) \rangle, P_{29}
\]
\[
\Omega_{\text{light}} \vdash \langle 29, \text{FTA}(29) \rangle, \text{i}0\text{fl}[+10] \cdot \text{i}0\text{oad}[4] \cdot \text{CS}_{32}, \{0, \ldots, 27\}, \langle P_{29}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_{29}, S_{20} \rangle \quad (5.2.6b)
\]

where \(P_{29} = P_0 \cap \{39 \mapsto \text{FTA}(39)\}\).

\[(A.2.16)\]
\[
\Omega_{\text{light}} \vdash \langle 32, \text{FTA}(32) \rangle, \text{i}0\text{oad}[4], \langle P_{29}, S_{20} \rangle \rightarrow \langle 34, \text{FTA}(34) \rangle, P_{29}
\]
\[
\Omega_{\text{light}} \vdash \langle 32, \text{FTA}(32) \rangle, \text{i}0\text{oad}[4] \cdot \text{i}0\text{store}[2] \cdot \text{CS}_{34}, \{0, \ldots, 29\}, \langle P_{29}, S_{20} \rangle \Rightarrow \text{PPS}, \langle P_{29}, S_{20} \rangle \quad (5.2.6b)
\]
A.2. LIGHTWEIGHT VERIFICATION PROOF

(A.2.17)
\[ \Omega_{\text{light}} \vdash \langle 34, \text{FTA}(34) \rangle, \text{istore}[2], \langle P_{29}, S_{20} \rangle \to \langle 36, \text{FTA}(36) \rangle, P_{29} \quad (5.3.3a) \]

(A.2.18)
\[ \Omega_{\text{light}} \vdash \langle 34, \text{FTA}(34) \rangle, \text{istore}[2], \langle P_{29}, S_{20} \rangle \to \text{PPS}, \langle P_{0}, S_{20} \rangle \quad (5.2.6b) \]

(A.2.19)
\[ \Omega_{\text{light}} \vdash \langle 39, \text{FTA}(39) \rangle, \text{iload}[4], \langle P_{29}, S_{20} \rangle \to \langle 41, \text{FTA}(41) \rangle, P_{0} \quad (5.3.3a) \]

(A.2.20)
\[ \Omega_{\text{light}} \vdash \langle 39, \text{FTA}(39) \rangle, \text{iload}[4], \text{ifne}[+6], \langle P_{29}, S_{20} \rangle \to \text{PPS}, \langle P_{0}, S_{20} \rangle \quad (5.2.6b) \]

(A.2.21)
\[ \Omega_{\text{light}} \vdash \langle 41, \text{FTA}(41) \rangle, \text{ifne}[+6], \langle P_{0}, S_{20} \rangle \to \langle 44, \text{FTA}(44) \rangle, P_{41} \quad (5.3.8b) \]

(A.2.22)
\[ \Omega_{\text{light}} \vdash \langle 41, \text{FTA}(41) \rangle, \text{ifne}[+6], \text{iload}[2], \text{CS}_{44}, \langle 0, \ldots, 39 \rangle, \langle P_{41}, S_{20} \rangle \to \text{PPS}, \langle P_{0}, S_{20} \rangle \quad (5.2.6b) \]

(A.2.23)
\[ \Omega_{\text{light}} \vdash \langle 41, \text{FTA}(41) \rangle, \text{ifne}[+6], \text{iload}[2], \text{CS}_{44}, \langle 0, \ldots, 39 \rangle, \langle P_{41}, S_{20} \rangle \to \text{PPS}, \langle P_{0}, S_{20} \rangle \quad (5.2.6b) \]

where \( P_{41} = P_{0} \cap \{ 47 \to \text{FTA}(47) \} \) here and below.

(A.2.24)
\[ \Omega_{\text{light}} \vdash \langle 47, \text{FTA}(47) \rangle, \text{iload}[2], \langle P_{41}, S_{20} \rangle \to \langle 49, \text{FTA}(49) \rangle, P_{41} \quad (5.3.3a) \]

(A.2.25)
\[ \Omega_{\text{light}} \vdash \langle 47, \text{FTA}(47) \rangle, \text{iload}[2], \text{CS}_{47}, \langle 0, \ldots, 44 \rangle, \langle P_{41}, S_{20} \rangle \to \text{PPS}, \langle P_{0}, S_{20} \rangle \quad (5.2.6b) \]
(A.2.24)
\[
\begin{align*}
\Omega_{\text{light}} & \vdash \langle 49, \text{FTA}(49) \rangle, \text{istore}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 51, \text{FTA}(51) \rangle, P_0 \quad (5.3.3a) \\
\Omega_{\text{light}} & \vdash \langle 49, \text{FTA}(49) \rangle, \text{istore}[4] : \text{iload}[3] : \text{CS}_{51}, [0, \ldots, 47], \langle P_0, S_{20} \rangle \Rightarrow \text{pps}, \langle P_0, S_{20} \rangle 
\end{align*}
\]

(A.2.25)
\[
\begin{align*}
\Omega_{\text{light}} & \vdash \langle 51, \text{FTA}(51) \rangle, \text{iload}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 53, \text{FTA}(53) \rangle, P_0 \quad (5.3.3a) \\
\Omega_{\text{light}} & \vdash \langle 51, \text{FTA}(51) \rangle, \text{iload}[3] : \text{istore}[2] : \text{CS}_{53}, [0, \ldots, 49], \langle P_0, S_{20} \rangle \Rightarrow \text{pps}, \langle P_0, S_{20} \rangle 
\end{align*}
\]

(A.2.26)
\[
\begin{align*}
\Omega_{\text{light}} & \vdash \langle 53, \text{FTA}(53) \rangle, \text{iload}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 55, \text{FTA}(55) \rangle, P_0 \quad (5.3.3a) \\
\Omega_{\text{light}} & \vdash \langle 53, \text{FTA}(53) \rangle, \text{iload}[4] : \text{istore}[3] : \text{CS}_{55}, [0, \ldots, 53], \langle P_0, S_{20} \rangle \Rightarrow \text{pps}, \langle P_0, S_{20} \rangle 
\end{align*}
\]

(A.2.27)
\[
\begin{align*}
\Omega_{\text{light}} & \vdash \langle 55, \text{FTA}(55) \rangle, \text{iload}[4], \langle P_0, S_{20} \rangle \rightarrow \langle 57, \text{FTA}(57) \rangle, P_0 \quad (5.3.3a) \\
\Omega_{\text{light}} & \vdash \langle 55, \text{FTA}(55) \rangle, \text{iload}[4] : \text{istore}[3] : \text{CS}_{57}, [0, \ldots, 53], \langle P_0, S_{20} \rangle \Rightarrow \text{pps}, \langle P_0, S_{20} \rangle 
\end{align*}
\]

(A.2.28)
\[
\begin{align*}
\Omega_{\text{light}} & \vdash \langle 57, \text{FTA}(57) \rangle, \text{istore}[3], \langle P_0, S_{20} \rangle \rightarrow \langle 59, \text{FTA}(59) \rangle, P_0 \quad (5.3.3a) \\
\Omega_{\text{light}} & \vdash \langle 57, \text{FTA}(57) \rangle, \text{istore}[3] : \text{goto}[-39] : \varepsilon, [0, \ldots, 55], \langle P_0, S_{20} \rangle \Rightarrow \text{pps}, \langle P_0, S_{20} \rangle 
\end{align*}
\]

(A.2.29)
\[
\begin{align*}
\Omega_{\text{light}} & \vdash \langle 59, \text{FTA}(59) \rangle, \text{goto}[-39], \langle P_0, S_{20} \rangle \rightarrow \langle 62, \top \rangle, P_0 \quad (5.3.9a) \\
\Omega_{\text{light}} & \vdash \langle 59, \text{FTA}(59) \rangle, \text{goto}[-39], [0, \ldots, 57], \langle P_0, S_{20} \rangle \Rightarrow \text{pps}, \langle P_0, S_{20} \rangle 
\end{align*}
\]

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. LIGHTWEIGHT CERTIFICATION PROOF

\[ \Omega \vdash 0, \text{aload}[1], \langle P_0, \emptyset \rangle \rightarrow 2, \text{FTA}(2), \langle P_0, \emptyset \rangle \]  

(A.3.1)

\[ \Omega \vdash 0, \text{aload}[1] \cdot \text{invokevirtual}[1] \cdot \text{CS}_2, \langle P_0, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, [20] \rangle \]  

(A.3.2)

\[ \text{(6.3.3a)} \]

\[ \text{(6.2.9b)} \]

\[ \text{(6.4.4c)} \]

\[ \text{(6.4.6b)} \]

\[ \text{(6.4.4b)} \]

\[ \text{(6.4.5b)} \]

\[ \text{(6.4.6b)} \]

\[ \text{(6.4.4b)} \]

\[ \text{(6.4.6b)} \]

\[ \text{(6.3.3a)} \]

\[ \text{(6.2.9b)} \]

where \( P_2 = P_0 \cap \{10 \mapsto \text{FTA}(10)\} \) here and below, and with \( \Theta_{\text{light}} = (\Omega, \text{LT}_2, \text{false}) \) with the contained list of possible exceptions obtained from \( \{E | E \leq \text{CS}_2 \rightarrow \text{Thrownable} \} \), and with \( \text{FTA}(10) = \langle \text{ST}_{10}, \text{LT}_{10} \rangle \).

\[ \Omega \vdash 5, \text{istore}[2], \langle P_2, \emptyset \rangle \rightarrow 7, \text{FTA}(7), \langle P_2, \emptyset \rangle \]  

(A.3.3)

\[ \Omega \vdash 5, \text{istore}[2] \cdot \text{goto}[-8] \cdot \text{CS}_7, \langle 0, 2 \rangle, \langle P_2, \emptyset \rangle \Rightarrow \text{PPS}, \langle P_0, [20] \rangle \]  

(A.3.4)

\[ \text{(6.3.9b)} \]

\[ \text{(6.2.9b)} \]

where \( P_7 = P_2 \cap \{15 \mapsto \text{FTA}(15)\} \) here and below.
where $P_{10} = P_7 \cap \{10 \rightarrow T\}$.

where $LT_{11}$ is defined by $FTA(11) = (\_, LT_{11})$.

where $LT_{14}$ is defined by $FTA(14) = (\_, LT_{14})$.

where $LT_{15}$ is defined by $FTA(15) = (\_, LT_{15})$. 
A.3. LIGHTWEIGHT CERTIFICATION PROOF

(A.3.10)  \[
\begin{align*}
\Omega & \vdash 20, \text{iload}[2], \langle P_0, \{20\} \rangle \rightarrow 22, \text{FTA}(22), \langle P_0, \emptyset \rangle \\
\end{align*}
\]

\[A.3.11\]
\[
\Omega \vdash 20, \text{iload}[2] \cdot \text{iload}[3] \cdot \text{CS}_{22}, \langle 0, \ldots, 18 \rangle, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle
\]

(A.3.11)  \[
\begin{align*}
\Omega & \vdash 22, \text{iload}[3], \langle P_0, \{20\} \rangle \rightarrow 24, \text{FTA}(24), \langle P_0, \emptyset \rangle \\
\end{align*}
\]

\[A.3.12\]
\[
\Omega \vdash 22, \text{iload}[3] \cdot \text{isub} \cdot \text{CS}_{24}, \langle 0, \ldots, 20 \rangle, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle
\]

(A.3.12)  \[
\begin{align*}
\Omega & \vdash 24, \text{isub}, \langle P_0, \{20\} \rangle \rightarrow 25, \text{FTA}(25), \langle P_0, \emptyset \rangle \\
\end{align*}
\]

\[A.3.13\]
\[
\Omega \vdash 24, \text{isub} \cdot \text{istore}[4] \cdot \text{CS}_{25}, \langle 0, \ldots, 22 \rangle, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle
\]

(A.3.13)  \[
\begin{align*}
\Omega & \vdash 25, \text{istore}[4], \langle P_0, \{20\} \rangle \rightarrow 27, \text{FTA}(27), \langle P_0, \emptyset \rangle \\
\end{align*}
\]

\[A.3.14\]
\[
\Omega \vdash 25, \text{istore}[4] \cdot \text{iload}[3] \cdot \text{CS}_{27}, \langle 0, \ldots, 24 \rangle, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle
\]

(A.3.14)  \[
\begin{align*}
\Omega & \vdash 27, \text{iload}[3], \langle P_0, \{20\} \rangle \rightarrow 29, \text{FTA}(29), \langle P_0, \emptyset \rangle \\
\end{align*}
\]

\[A.3.15\]
\[
\Omega \vdash 27, \text{iload}[3] \cdot \text{ifle} [+ 10] \cdot \text{CS}_{29}, \langle 0, \ldots, 25 \rangle, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle
\]

(A.3.15)  \[
\begin{align*}
\Omega & \vdash 29, \text{ifle} [+ 10], \langle P_0, \{20\} \rangle \rightarrow 32, \text{FTA}(32), \langle P_{29}, \emptyset \rangle \\
\end{align*}
\]

\[A.3.16\]
\[
\Omega \vdash 29, \text{ifle} [+ 10] \cdot \text{iload}[4] \cdot \text{CS}_{32}, \langle 0, \ldots, 27 \rangle, \langle P_{29}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_{29}, \{20\} \rangle
\]

where \( P_{29} = P_0 \cap \{39 \mapsto \text{FTA}(39)\} \).

(A.3.16)  \[
\begin{align*}
\Omega & \vdash 32, \text{iload}[4], \langle P_{29}, \{20\} \rangle \rightarrow 34, \text{FTA}(34), \langle P_{29}, \emptyset \rangle \\
\end{align*}
\]

\[A.3.17\]
\[
\Omega \vdash 32, \text{iload}[4] \cdot \text{istore}[2] \cdot \text{CS}_{34}, \langle 0, \ldots, 29 \rangle, \langle P_{29}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_{29}, \{20\} \rangle
\]
where $P_{41} = P_0 \cap \{47 \mapsto \text{FTA}(47)\}$ here and below.

(A.3.21)

$\omega \vdash 44, \text{iload}[2], \langle P_{41}, \{20\} \rangle \rightarrow 46, \text{FTA}(46), \langle P_{41}, \{20\} \rangle$ (6.3.3a)

$\omega \vdash 44, \text{iload}[2] \cdot \text{ireturn} \cdot \text{CS}_{46}, \{0, \ldots, 41\}, \langle P_{41}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$ (6.2.9b)

(A.3.22)

$\omega \vdash 46, \text{ireturn}, \langle P_{41}, \{20\} \rangle \rightarrow \langle 47, \top \rangle, \langle P_{41}, \{20\} \rangle$ (6.3.11b)

$\omega \vdash 46, \text{ireturn} \cdot \text{iload}[2] \cdot \text{CS}_{47}, \{0, \ldots, 44\}, \langle P_{41}, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$ (6.2.9b)

(A.3.23)

$\omega \vdash 47, \text{iload}[2], \langle P_{41}, \{20\} \rangle \rightarrow 49, \text{FTA}(49), \langle P_0, \{20\} \rangle$ (6.3.3a)

$\omega \vdash 47, \text{iload}[2] \cdot \text{istore}[4] \cdot \text{CS}_{49}, \{0, \ldots, 46\}, \langle P_0, \{20\} \rangle \Rightarrow \text{PPS}, \langle P_0, \{20\} \rangle$ (6.2.9b)
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).

```java
// ON—CARD CLASSES.

// Thrown when attempting to read uninitialized credit card.
```
class UnsetCrCard extends Exception {

    // On—card class with credit card information.
    class CrCardRd {
        int it; // credit card number (0 if uninitialized)
        public int getIt() throws UnsetCrCard {
            if (it == 0) throw new UnsetCrCard();
            return it;
        }
    }

    // Thrown to bail out when credit card cannot be read.
    class Abort extends Exception {

        // Check sum interface.
        interface CkSum {
            public int cksum(CrCardRd ccnum) throws Abort;
        }

        // DOWNLOADED CLASSES.

        // Implementation of check sum calculation.
        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; }
                }
            }

        }

    }

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
234  

APPENDIX A. CKSUM() EXAMPLE DETAILS

Method Gcd11()

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 =
A.5. EXECUTION OF THE LBV PROGRAM ON GCD11

- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>\) 
  0: aload 1 
- \(<\text{LCrCardRd};, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  1: invokevirtual methodref(\(\text{LCrCardRd};, \text{methsig}(\text{getIt}, [\]), I) 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  4: istore 2 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  5: goto +12 
- \(<\text{LUnsetCrCard};, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  8: pop 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  9: new \text{LAbort}; 
- \(<\text{LAbort};, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  12: dup 
- \(<\text{LAbort};, \text{LAbort};, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  13: invokespecial methodref(\(\text{LAbort};, \text{methsig}(\text{<init}}, [\]), V) 
- \(<\text{LAbort};, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  16: athrow 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  17: iconst 0 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  19: istore 3 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  20: iload 2 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  21: iload 3 
- \(<I.I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  22: isub 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  23: istore 4 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  25: iload 4 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  27: ifle +9 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  29: iload 4 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  30: iload 4 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  32: istore 2 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  33: goto -13 
- \(<{\star}, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  36: iload 4 
- \(<I, \text{LCgcd11}; \text{LCrCardRd}; \text{.bot}.\text{bot}.\text{bot}>) 
  38: ifne +5
APPENDIX A. CKSUM() EXAMPLE DETAILS

- <*, 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 \( \text{ClassHier} \vdash_{bv} \overrightarrow{\text{Type}_\bot} \sqsubseteq \overrightarrow{\text{Type}_\bot} \) ........................................ 53  
4.1.28 \( \text{ClassHier} \vdash_{bv} (\text{FrameType}_{\text{MS,ML}}) \times \sqsubseteq_{\text{MS,ML}} (\text{FrameType}_{\text{MS,ML}}) \times \) .......................... 59  
4.1.30 \( \text{ClassHier} \vdash_{bv} \text{StackType}_{\text{MS}} \sqsubseteq_{\text{MS}} \text{StackType}_{\text{MS}} \) ........................................ 60  
4.1.30 \( \text{ClassHier} \vdash_{bv} \text{LocalType}_{\text{ML}} \sqsubseteq_{\text{ML}} \text{LocalType}_{\text{ML}} \) ........................................ 60  
4.1.40 \( \text{ClassHier} \vdash_{bv} \text{Propagate} \times \text{ClassIdent} \times \text{ExcAtt} \) ........................................ 88  
4.2.7 \( \text{StdContext} \vdash_{bv} \text{Method}, \text{FrameTypeAssign} \) ........................................ 65  
4.2.8 \( \text{CodeContext} \vdash_{bv} \text{PPoint}, \text{Code}, \text{PPoints}_{\text{tsafe}} \Rightarrow \text{PPoints} \) ....................... 66  
4.3.3 \( \text{CodeContext} \vdash_{bv} \text{PPoint} \times \text{Ins}_{\text{tsafe}} \Rightarrow \text{PPoint} \) ............................................... 68  
4.4.11 \( \text{ExcContext} \vdash_{bv} \text{PPoint}, \text{Excs} \) ........................................ 85  
4.4.19 \( \text{ClassHier} \vdash_{bv} \text{Propagate}, \text{ClassIdent}, \text{ExcAtt} \) ........................................ 88  
4.4.20 \( \text{ClassHier} \vdash_{bv} \text{ClassIdent}, \text{ExcAtt} \) ........................................ 88  

5.2.5 \( \text{StdContext} \vdash_{lbv} \text{Method}, \text{Cert} \) ................................................................. 107  
5.2.6 \( \text{LightContext} \vdash_{lbv} \text{CodeStat}, \text{CodeSeq}, \text{PPoints}, \text{DelayConstr}_{\text{tsafe}} \Rightarrow \text{PPoints}, \text{DelayConstr} \) .... 108  
5.3.1 \( \text{LightContext} \vdash_{lbv} \text{CodeStat} : \text{Ins}, \text{DelayConstr}_{\text{tsafe}} \Rightarrow \text{CodeStat}, \text{Pending} \) ............... 110  
5.4.4 \( \text{ExcContext} \vdash_{lbv} \text{PPoint}, \text{Excs} \) ................................................................. 117  

6.2.4 \( \text{StdContext} \vdash_{lbc} \text{Method}, \text{FrameTypeApprox}, \text{Cert} \) ........................................ 127  
6.2.9 \( \text{CodeContext} \vdash_{lbc} \text{PPoint}, \text{CodeSeq}, \text{PPoints}, \text{PCert}_{\text{certify}} \Rightarrow \text{PPoints}, \text{PCert} \) ........... 129  
6.3.1 \( \text{CodeContext} \vdash_{lbc} \text{PPoint} : \text{Ins}, \text{PreConstr}_{\text{certify}} \Rightarrow \text{PPoint}, \text{FrameType}, \text{PreConstr} \) ........... 133  
6.4.3 \( \text{ExcContext} \vdash_{lbc} \text{PPoint}, \text{Excs}, \text{ExcHandlers}, \text{PreConstr}_{\text{certify}} \Rightarrow \text{PreConstr} \) ............. 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Definition/Notation Remark/Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.11</td>
<td>Definition (Array Instructions)</td>
</tr>
<tr>
<td>3.1.12</td>
<td>Remark (Formalization of newarray)</td>
</tr>
<tr>
<td>3.1.13</td>
<td>Definition (Constant Pool Instructions)</td>
</tr>
<tr>
<td>3.1.14</td>
<td>Remark (Formalization of new)</td>
</tr>
<tr>
<td>3.1.15</td>
<td>Definition (The Branch Instructions)</td>
</tr>
<tr>
<td>3.1.16</td>
<td>Definition (The Goto Instruction)</td>
</tr>
<tr>
<td>3.1.17</td>
<td>Definition (The athrow Instruction)</td>
</tr>
<tr>
<td>3.1.18</td>
<td>Definition (Return Instructions)</td>
</tr>
<tr>
<td>3.1.19</td>
<td>Remark (The Java Card and the J2ME instruction sets)</td>
</tr>
<tr>
<td>3.1.20</td>
<td>Example (The checksum bytecode representation.)</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Definition (The Formal Verification Context)</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Definition (Constant Pool Items)</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Notation (The class hierarchy subscript)</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Example (The checksum class file verification)</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Definition (A Method)</td>
</tr>
<tr>
<td>3.2.8</td>
<td>Definition (Bytecode Formalization)</td>
</tr>
<tr>
<td>3.2.10</td>
<td>Definition (Bytecode Sequence Program Points)</td>
</tr>
<tr>
<td>3.2.12</td>
<td>Definition (The Exception Handler Table)</td>
</tr>
<tr>
<td>3.2.14</td>
<td>Example (The checksum method formalization)</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Definition (The Class Hierarchy)</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Lemma (The subtype semi-lattice property)</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Example (The checksum class hierarchy formalization)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Definition/Notation Remark/Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1</td>
<td>Example (Type safety for method invocation)</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Example (Type safety for object assignment)</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Observation (Type safety for dynamic type assignment)</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Definition (General Verification Constraints)</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Definition (Type assignment compatibility)</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Definition (The Abstracted Type Sort)</td>
</tr>
<tr>
<td>4.1.10</td>
<td>Definition (Type Assignment Compatibility)</td>
</tr>
<tr>
<td>4.1.11</td>
<td>Definition (The Type Assignment Compatibility Relation)</td>
</tr>
<tr>
<td>4.1.12</td>
<td>Lemma (Type compatibility as a partial order)</td>
</tr>
<tr>
<td>4.1.13</td>
<td>Example (The checksum compatibility ordering)</td>
</tr>
<tr>
<td>4.1.14</td>
<td>Proposition (The extended type lattice)</td>
</tr>
<tr>
<td>4.1.15</td>
<td>Remark (Multi-dimensional arrays)</td>
</tr>
<tr>
<td>4.1.16</td>
<td>Remark (Interfaces)</td>
</tr>
<tr>
<td>4.1.18</td>
<td>Definition (Frame Types)</td>
</tr>
<tr>
<td>4.1.19</td>
<td>Notation (Stack types)</td>
</tr>
<tr>
<td>4.1.20</td>
<td>Notation (Local types)</td>
</tr>
<tr>
<td>4.1.21</td>
<td>Definition (Stack Type Size)</td>
</tr>
<tr>
<td>4.1.22</td>
<td>Definition (Local Type Size)</td>
</tr>
<tr>
<td>4.1.24</td>
<td>Definition (Bounded Frame Types)</td>
</tr>
<tr>
<td>4.1.25</td>
<td>Definition (The Frame Type Sort)</td>
</tr>
<tr>
<td>4.1.27</td>
<td>Remark (The StackType)</td>
</tr>
</tbody>
</table>
LIST OF DEFINITIONS, ETC.

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 `cksum()` 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 `cksum()` method bytecode. ............................... 40
3.8 Formalizing checksum and its verification context. .......... 48

4.1 A general `(Type₁, ⊑_{CH})` ordering .......................... 54
4.2 The `(Type₁, ⊑_{CH+k})` example ordering ...................... 56
4.3 An unsafe `cksum()` variant. .................................. 81
4.4 Type safety for exceptions raised at PP. ....................... 85
4.5 A `cksum()` frame type assignment. ............................ 90
4.6 Context for `cksum()` 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” `cksum()` 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
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Instruction rule correspondence</td>
<td>143</td>
</tr>
<tr>
<td>6.6</td>
<td>Exception rule type safety</td>
<td>147</td>
</tr>
<tr>
<td>6.7</td>
<td>Exception certification updates</td>
<td>147</td>
</tr>
<tr>
<td>6.8</td>
<td>Exception lightweight updates and type safety</td>
<td>148</td>
</tr>
<tr>
<td>6.9</td>
<td>No-catch exception-rule correspondence</td>
<td>149</td>
</tr>
<tr>
<td>6.10</td>
<td>Catch exception-rule correspondance</td>
<td>149</td>
</tr>
<tr>
<td>7.1</td>
<td>Sorts implemented by basic Java types</td>
<td>151</td>
</tr>
<tr>
<td>8.1</td>
<td><code>cksum()</code> that verifies with Leroy’s algorithm</td>
<td>204</td>
</tr>
</tbody>
</table>
Bibliography


[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


Index

aaload, 34, 72
aastore, 34, 72
abrupt instructions, 79
abstract interpretation, 17
aconstnull, 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
⊥ [], 72
⊥, 52
⊥ [], 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
INDEX

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
Gcd, 21
General Verification Constraints, 51
getfield, 35, 36, 74
goto, 35, 37, 77, 113
GotoIns, 35
greatest lower bound, 26
ι, 31
iadd, 32, 33, 69
iload, 34, 72
iastore, 34, 72
iconst₀, 32, 33, 69
iconst₁, 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
ublisher, 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
ldcw, 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
ublisher, 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
Ωlight, 106
OP, 31
OpCode, 31
implementation, 151
operational, 16
operator signature, 25
order
  partial, 26
  \subseteq_{MS, ML}, 59
  \subseteq_{MS, ML}, 62
  \subseteq_{ML}, 60
  \subseteq_{CH}, 62
  \subseteq_{MS}, 60
  \subseteq_{CH}, 62
  \subseteq_{CH}, 55

P, 98
P_0, 103
partial function, 25
partial order, 26, 55, type compatibility62
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
PPPoints, 44
PPS, 44
PPSC, 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
\tau, 53
\tau_{aref}, 53
\tau_{ob}, 53
\Theta, 84
\Theta_{light}, 116
ThrowIns, 35
Τ, 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