Technische Universität Dresden

Fakultät Informatik

Diplomarbeit

zum Thema

Coalgebraic Representation of the Java Card Virtual Machine in PVS

von

Frank Felfe

geboren am 23. April 1975 in Dresden

Eingereicht am 21. Februar 2001

Betreuer:	Dr. B. Jacobs
	Dep. Sci., Univ. Nijmegen
Verantwortlicher Hochschulleiter:	Prof. Dr. rer. nat. habil. H. Reichel

Contents

List of Figures

1. Introduction

During the past years Java Card has become a widely accepted standard for smart card programming and programming of other small, resource-constrained embedded devices. Java Card applications can be developed and tested on a Java environment and then converted to a form which can be installed onto a Java Card device making it a convenient and fast development platform.

The Java Card application is executed on the smart card by the Java Card virtual machine. Only the results of this computation are reported to the cards environment making it much more secure than conventional card devices. The Java Card virtual machine is part of the Java Card platform and its operational semantics are described in the Java Card virtual machine specification [3] by Sun Microsystems in a rather informal way.

Since smart cards are used for authentication and other critical tasks, security and safety of applications running on smart cards are a major issue for usability and acceptance of this technology. There are several projects aimed at developing methods and tools for verifying Java programs and object-oriented programming language in general. While these tools could be used to verify Java Card programs they only operate on the source code.

In order to verify the application that is actually installed onto the device one has to apply tools that investigate the Java Card bytecode directly. Further significant differences in the structure of Java bytecode and Java Card bytecode make it difficult to use tools that were designed for verification of Java bytecode.

In this thesis I develop a way to formalize the Java Card virtual machine including a representation of the CAP file which contains the class structure, a state-based memory model and a representative part of the operational semantics of the Java Card virtual machine.

In contrast to other formalizations I will work out the coalgebraic type structure inhabited in the virtual machine and use these to describe the Java Card virtual machine. The access of the machine memory is abstracted from the memory model and the operational semantics which will make the formalization more flexible and adaptable to actual implementations. Then I will verify example bytecode programs that I had to write by hand, since the formalization lacks some important components to deal with real Java

Card bytecode. For the formalizations and verifications I use the PVS tool.

The CAP file and the machine memory representation in this thesis will only cover those parts needed by the Java Card virtual machine. A complete representation of the Java Card platform would also require a formalization of the Java Card Runtime Environment [4] and the Java Card Application Programming Interface [5] which is beyond the scope of this thesis.

This thesis is structured as follows. Section 2 introduces preliminaries that are necessary to understand this work. Chapter 3 describes the representation of the CAP file that is loaded onto the card. In Section 4 I develop a memory model for the state of the Java Card virtual machine. Section 5 will describe the coalgebra type of the Java Card virtual machine and the formalization of the operational semantics. Section 6 presents some example programs and small verification examples to illustrate the use of the formalization. A deeper discussion of how such a formalization can be used for verification of Java Card applications is not a topic of this thesis. Section 7 summarizes the results and outlines future work.

2. Preliminaries

This chapter presents all technical and mathematical foundations that are necessary for understanding this work. Basic knowledge about the Java programming language, object oriented programming and higher order logic is recommended but not required to understand this work.

In section 2.1 I describe basic concepts the Java Card platform and smart card technology is based on. Section 2.2 gives some backgrounds and motivations for software verification. The PVS tool that is used for this work is described briefly and related work is introduced. A type theory language that is used throughout this work is introduced and described in section 2.3.

2.1. Java Card Technology

Java Card is a programming language and development platform for writing programs for smart cards. *Smart cards* are small plastic cards with the size of a credit card which can store and execute programs right on it. The smart card that is made for Java Card programs contains a small computer called Java Card virtual machine that executes the programs which are in a special format called Java Card *bytecode*.

The *Java Card* language was created by Sun Microsystems, Inc. It is a proper subset of the Java language, defined in the Java Language Specification [1]. The Java Card platform combines a complete language API defined in the Java Card Application Programming Interface [5] and a runtime environment. The Java Card runtime environment (JCRE) is defined in the Java Card Runtime Environment Specification [4]. The JCRE contains the Java Card virtual machine (JCVM) defined in the Java Card Virtual Machine Specification [3]. The JCVM is similar to the Java virtual machine (JVM) which is defined in the Java Virtual Machine Specification [2]. This work is based on the specifications for Java Card version 2.1.1. When speaking about the JCVM specification the Java Card 2.1.1 Virtual Machine Specification [3] is meant. JCRE specification refers to the Java Card 2.1.1 Runtime Environment Specification [4].

Java Card is an object oriented programming language in C-style notation and is optimized for use in smart cards and other resource restricted and embedded devices. While Java Card supports most of Javas object oriented features it does not support features like threading, floating-point and string data-types which are not needed in smart cards. A security system based on separated contexts that are assigned to objects and methods was added to the JCVM to enforce additional security needed in smart card environments.

The complete definition of the Java Card API and the JCRE plus the fact that Java Card is a subset of the popular Java language have made it a convenient and popular development platform. The fact that smart cards not only contain data but also the programs to process the data makes them a powerful extension of classical card technology used in bank or telephone cards. Unlike conservative card types they can operate independent from networks and external data. Thus, they solve several problems of classic cards and create several new ones putting security at stake. These unique features make smart cards useful in environments where conservative technologies are not applicable.

2.1.1. Smart Card Applet Development

Figure 2.1 describes the development process of a Java Card Program which is described in the JCVM specification [3]. Since Java Card is a subset of the Java language Java Card

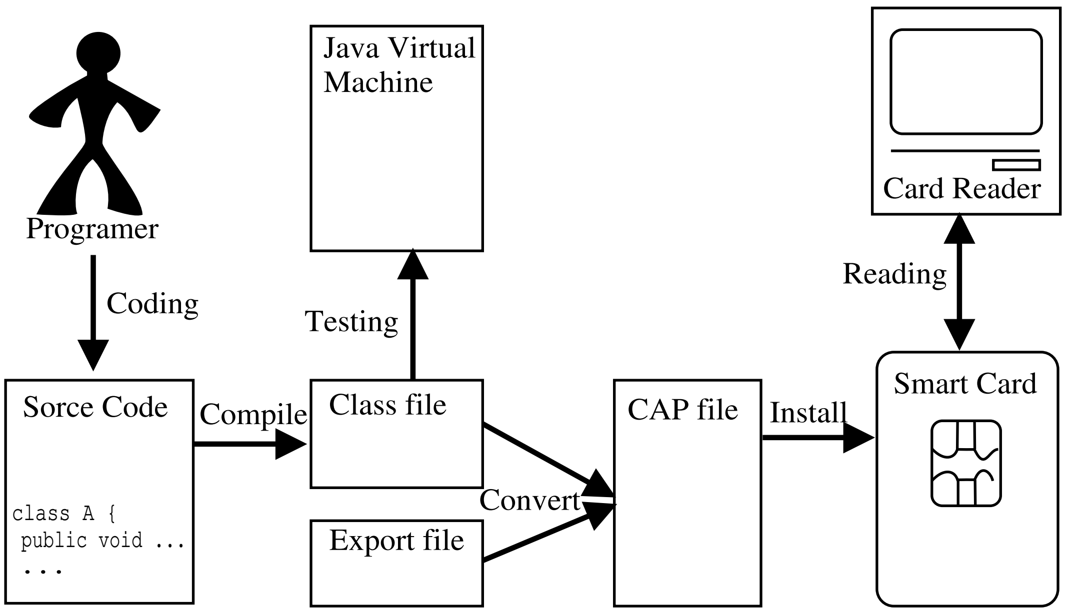

Figure 2.1.: Smart Card applet development

programs can be written as Java programs. A programmer already familiar with Java programming does not need to learn a new language or get used to a new development environment.

The Java Card source code is compiled into a Java *class file* with a Java compiler. The Java class file is the binary format of the Java language and contains the complete class structure from the compiled source code. Java bytecode is not executed by a native machine but instead by the JVM which can be implemented for any platform. The Java Card binaries in form of the Java class file can thus be run and tested with a JVM which usually is contained in a Java development environment.

The class files are then converted into a *CAP file* using an applet converter. Several class files can be compiled into one CAP file, thus the CAP file can contain several programs. CAP is an abbreviation for converted applet and the CAP file is also called Java Card *package*. The programs in a CAP file are called *applets*. Since Java Card applets are run on resource restricted devices the CAP file is optimized in size. While it still contains the complete class structure that is necessary for execution a lot of data contained in the class file has been eliminated. Additionally the data in the CAP file has been restructured to better fit the JCVM.

Export files contain name and link information for the contents of other packages that are imported by the converted classes. One or more export files may be necessary for creating a CAP file. Export files are produced when an applet is converted.

The CAP file is then installed onto the smart card with an installation tool. It is possible to install several CAP files onto one smart card and add new CAP files at any time provided there is still enough space on the smart card. The process of applet linking is performed on the smart card in order to not let data about the CAP files emerge from the smart card and grant anonymity of the smart card contents.

In contrast to the JVM no bytecode verifier is defined for the JCVM. The JCVM depends on the compiler to guarantee the safety of the bytecode execution. The *bytecode verifier* of the JVM is used to verify if the bytecode that is executed in the JVM is well formed and thus safe to execute. The bytecode verifier is run before the bytecode is linked and executed. Also the JCVM does not define dynamic class linking which is not necessary since all accessible CAP files reside on the smart card and have already been linked during installation.

When the CAP file is installed and linked, the card is ready for use. Smart cards are used with a *smart card reader* that is an electronic device that can order the card to start an applet, input data into the card that is requested for the computation and receive the results of the computation. Data stored on the smart card and information about the applets can not be read from the card. Once an applet has been installed there is no regular way to extract the card contents, making it in theory a secure technology.

2.1.2. Logic Structure of a Smart Card

On every smart card resides a small computer. This computer is specified as JCRE in [4]. The logic structure of a smart card is shown in figure 2.2. The JCRE is able to communicate with the environment of the smart card. So the applet installation device can load a CAP file onto the card and order the card to install and link the CAP file. The process of applet linking and installation is performed by a part of the JCRE that is called the applet installer. The CAP file is then written to the applet memory.

The smart card reader can order the JCRE to run an applet. The JCRE then executes the applets applet initialization method which does the necessary initialization of the JCVM and the JCVM memory which includes determining the class and the main method that belong to the applet and creating a class instance and method instance in the JCVM memory for these. In the JCVM formalization the JCVM memory will be referred as JCVM *state* according to the usual notion of abstract automatons.

The applet initialization method and the applet are executed by the JCVM which is also part of the JCRE. The JCVM reads data from the class structure in the CAP file and data from the JCVM memory. Then it updates the memory according to the semantics specification of the next instruction which is determined from the JCVM state. Additionally the JCVM can communicate via the JCRE with the card reader and receive input data and transmit computation results.

2.2. Software Verification

The idea of formal software verification is to build a representation of the software, i.e. its functionality, and investigate this representation using formal methods. By doing this, properties of the investigated software can be shown and proven formally and errors can be determined.

Testing and verification of the functionality of products is part of the development process of almost any product, and software development is no different. There are essentially 3 levels in which software verification can take place:

- Basic concepts of software implementations, mechanisms and algorithms can be specified and verified. On this level the proper functioning of the design of software is investigated by using methods of the so called algebraic specification which is widely studied. Design verification is a common technique for the verification of software products nowadays. While conceptional errors can be detected this way, no statement about the actual implementation can be made.

- By creating a formal specification of the source code it is possible to formally investigate the actual implementation of the software. This method is often used in

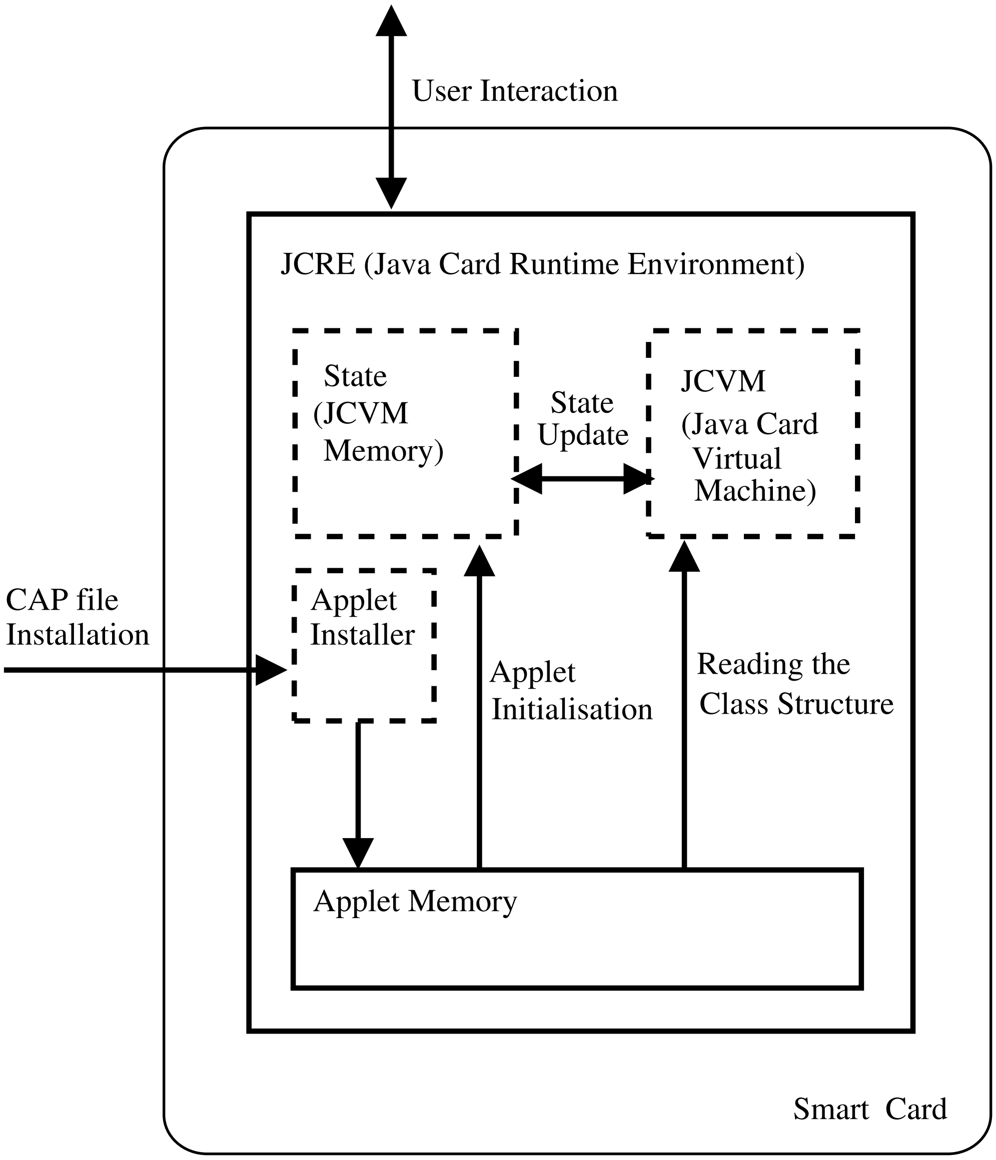

Figure 2.2.: The logic structure of a smart card

combination with the first method. The semantics of programming languages that are used for software development, however, is very complex. Especially object oriented languages, that are the standard in commercial software development, can not be modeled with traditional techniques. The use of coalgebraic methods to model object semantics has led to new approaches in source code verification. Today several projects try to investigate and use the possibilities of verifying source code using formal methods. However, source code still depends on the proper definition on the semantics of programming languages and the implementation of the compiler to work correctly.

- Thus, it would be most efficient to verify the correctness of the machine-code that is executed by the computer. This implies two problems: The machine-code is usually far more complex in size than the source-code, thus requiring exceptionally more computing power. Further such verifications would only hold for a particular platform for which a formalization would be required.

  The establishment of the Java language which is executed by a platform independent virtual machine has inspired researchers to investigate this field and use the Java bytecode as basis for verifications. Additional advantages of Java are that the specification of the JVM is reasonably simple and completely available as an open standard.

2.2.1. Related Work

A quantity of work to formalize virtual machines has been done. Cohen [24] formalized the so called *defensive* JVM in ACL2. A defensive virtual machine performs runtime checks to guaranty a type-safe execution of the bytecode.

A *non-defensive* version of the JVM has been formalized by Pusch [6] in Isabelle/HOL. Such a JVM does not perform any runtime checks. The type-safety of the execution must be verified using a bytecode verifier. Pusch formalized the bytecode verifier in Isabelle/HOL too [7] and showed in a formal proof that it is safe to execute bytecode if it has been certified by the bytecode verifier.

Qian [8] has given a formal specification of the instructions of the JVM for which he also describes a static type inference system. He proves that runtime data is type-correct provided the JVM program is well typed. It turns out, that the treatment of subroutine calls is the most complex task in the bytecode verification process. The work is, however, done with paper and pencil instead of utilizing a theorem prover.

Another formalization of the JVM is done by Bertelsen [23] where he describes the dynamic semantics of Java bytecode in the form of an operational semantics of the JVM.

The JCVM has been formalized by Barthe et al. [9] using the theorem prover Coq. They describe the JCVM in form of its operational semantics by giving a rule for each instruction describing the instructions effect on the state. While they claim to have formalized a complete set of JCVM instructions there is only a small description of this work from a workshop available.

Denny and Jensen [17] give a formal description of the transformation of the Java class file to the CAP file and then prove that this transformation preserves the class structure.

Leavens et al. [25] have developed a interface specification language for Java called JML. In [20] Poll, Berg and Jacobs use JML to specify all classes of the Java Card API and make implicit assumptions that inhabit the underlying implementation explicit. Such specifications are meant to increase the reliability of the code that the Java Card API is based on.

Russo et al. [22] described a large subset of the Java language including inheritance, dynamic linking and multi-threading as structural operational semantics. Additionally they derive an interactive programming environment from their specification. Nipkow and Oheimb [21] have formalized a subset of Java which they call BALI, together with its type system. Then they proved that BALI is type-save.

The use of terminal coalgebras to describe object semantics has been proposed by Reichel [15]. Terminal coalgebras are shown to be a suitable formal basis for object types in the sense of the object oriented programming paradigm. Jacobs,Tews and Rothe use this approach to develop the Coalgebraic Class Specification Language (CCSL) [16] which is aimed at coalgebraic specification of behavioral types or classes of object oriented languages. CCSL is part of the LOOP project [13].

The LOOP project tries to verify sequential Java programs. The underlying memory model of the LOOP project [10] was the basis for the development of the memory model of the JCVM in this work. An extensive description of the mechanisms used for reasoning about Java programs in higher order logic is given by Huisman [11], who has been working on the LOOP project.

2.2.2. The Prototype Verification System (PVS)

For the formal specification and proofs in this work I use the Prototype Verification System or PVS [19]. PVS is developed at the computer science laboratory at the Stanford Research Institute. It combines a specification language for formalizing specifications and a proof tool for carrying out proofs in these specifications. As the name suggests PVS is used to design prototypes of hardware and software systems. Abstract specifications can then be analyzed with formal proofs that makes it a suitable tool for verification of software and hardware.

PVS uses Allegro Lisp, a fast Common Lisp implementation, as back-end for representing theories and performing proofs. However, the user specifies theories in the PVS language which is intended to be intuitive and understandable without knowledge about special mathematical calculi. The PVS language is a functional language based on higher order logic.

The PVS prover is based on the sequent calculus. In addition to the standard proof rules the PVS prover offers a comfortable set of predefined proof strategies which make proofs very intuitive. Users also can define their own strategies using the underlying lisp system. More information and documentation about PVS as well as several examples of specifications and verifications made with the PVS tool can be found in [19].

For this work PVS Version 2.3 (patch level 1.2.2.54) was used in combination with XEmacs 20.4 under Debian GNU/Linux 2.2. The proofs for the examples in chapter 6 were all carried out on a Pentium MMX 233MHz with 96MB ram and the mentioned PVS/Linux combination unless specified differently.

2.3. A Simple Type Theory

While the PVS system is used for the formalization of the JCVM as well as for the proofs, I use an independent *type theory* to present the formalization in this thesis. This makes it more understandable for readers not familiar with the PVS language and easier to translate for use with different theorem provers. The type theory is an abstraction and simplification of the PVS language. The same type theory has been used by Jacobs in [10] and [12] and is described detailed by Huisman in [11].

Both languages are used to form formulas in higher order logic. The type theory is used to build types from:

- type variables $\alpha, \beta, \ldots$
- type constants like *nat*, *int*, *boolean*, etc.
- the recursive type constructor *list*
- exponent types $\sigma \rightarrow \tau$
- labeled product (or record) types $[lab_1 : \sigma_1, \ldots, lab_n : \sigma_n]$
- labeled coproduct (or variant) types $\{lab_1 : \sigma_1 \mid \ldots \mid lab_n : \sigma_n\}$

for given types $\sigma, \tau, \sigma_1 \ldots \sigma_n$. The label lab_i are pare-wise distinct. Terms of these types are called the inhabitants of the types. The relevant terms and operations for these types are presented now.

For type list the constructors *nil* and *cons* are defined as well as the destructors *head* and *tail.* head and tail are partial functions over type list which are only defined for lists unequal nil. The following equation holds.

– TYPE THEORY

$$\forall l\colon \mathsf{list}[\alpha].l \neq \mathsf{nil} \supset \mathsf{cons}(\mathsf{head}\, l, \mathsf{tail}\, l) = l$$

Please note that the partial definition of head and tail require to ensure that the list l is unequal nil before head or tail can be applied to l. The corresponding definition of head and tail shall be omitted here since in the PVS formalization a predefined type list is used and the PVS system cares for the necessary constraints. The type list is used to describe stacks for which it has all necessary properties.

The standard notation for lambda abstraction $\lambda x\colon \sigma.\, M$ and application MN is used for exponent types. This way lambda-terms can be built by binding typed variables.

An update operator *WITH* is defined for exponent types.

– TYPE THEORY

$$f\,\mathsf{WITH}\ [i = N]$$

If f inhabits type $\sigma \to \tau$, i inhabits type σ and N inhabits type τ then N is assigned to fi. fx for an x inhabiting type σ and $x \neq i$ remains unchanged. Thus, it is an abbreviation for the lambda-term

– TYPE THEORY

$$\lambda x\colon \sigma.\ \mathsf{IF}\ x = i\ \mathsf{THEN}\ N\ \mathsf{ELSE}\ fx$$

for which the following equations hold.

– TYPE THEORY

$$(f\,\mathsf{WITH}\ [i = N])i = N$$
$$(f\,\mathsf{WITH}\ [i = N])j = fj \vee i = j$$

Consider the labeled product $N = (lab_1 = M_1, \ldots, lab_n = M_n)$ of type $[lab_1\colon \sigma_1, \ldots, lab_n\colon \sigma_n]$. Then the labeled product N satisfies the following equations

– TYPE THEORY

$$(lab_1 = M_1, \ldots, lab_n = M_n).lab_i = M_i$$
$$(lab_1 = N.lab_1, \ldots, lab_n = N.lab_n) = N$$

that define the intuitive behavior of selection $N.lab_i$ and tupleing that is used to create labeled products. Labeled products are a more general descriptions of cartesian product types.

The update operator *WITH* is defined for labeled products M too,

– TYPE THEORY

$$M \ \mathsf{WITH}\ [lab_i = N]$$

which replaces $M.lab_i$ with N and satisfies the following intuitive equations.

– TYPE THEORY

$$\begin{aligned}&(M \ \mathsf{WITH}\ [lab_i = N]).lab_i = N\\&(M \ \mathsf{WITH}\ [lab_i = N]).lab_j = M.lab_j \vee i = j\end{aligned}$$

Given a term N of the coproduct type $\{lab_1 : \sigma_1, \dots , lab_1 : \sigma_1\}$ and terms $L_1 : \sigma_1, \dots , L_1 : \sigma_1$, the *CASES* term is defined to analyze the term N.

– TYPE THEORY

$$\mathsf{CASES}\ N\ \mathsf{OF}\ \big\{ \mid lab_1 x_1 \mapsto L_1(x_1) \mid \dots \mid lab_n x_n \mapsto L_n(x_n) \big\}$$

The term L_i that corresponds to the term N is returned.

Given an additional tagged term $lab_i M$ inhabiting the type of N where $M : \sigma_i$ the following equation holds.

– TYPE THEORY

$$\mathsf{CASES}\ lab_i M\ \mathsf{OF}\ \big\{ \mid lab_1 x_1 \mapsto L_1(x_1) \mid \dots \mid lab_n x_n \mapsto L_n(x_n) \big\} = L_i[M/x_i]$$

All free occurrences of x_i in the returned term L_i are replaced with the term M. The CASES-operator can be used to investigate labeled coproducts and define a resulting term depending an the type of the term inhabited in the labeled coproduct.

Properties of terms can be specified using first order *predicate logic* which is defined in the standard way. First order formulas are of type *boolean*. They can be constructed from the constants *true* and $false$, formula variables $\varphi, \psi, \dots$, the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication) and $\neg$ (negation) and the typed quantifiers $\forall x : \sigma.\varphi$ (all quantifier) and $\exists x : \sigma.\varphi$ (existential quantifier).

For terms M and N of the same type and a first order formula φ the conditional term $\mathsf{IF}\ \varphi\ \mathsf{THEN}\ M\ \mathsf{ELSE}\ N$ is defined which satisfies the following equations.

– TYPE THEORY

$$\mathsf{IF}\ true\ \mathsf{THEN}\ M\ \mathsf{ELSE}\ N = M$$
$$\mathsf{IF}\ false\ \mathsf{THEN}\ M\ \mathsf{ELSE}\ N = N$$
$$\mathsf{IF}\ \varphi\ \mathsf{THEN}\ L[true/z]\ \mathsf{ELSE}\ L[false/z] = L[\varphi/z]$$

The following conventions define some alias-types and macro-types to simplify the formalization and make it more readable. All of the following type definitions are used often throughout the formalization.

The following type constants denoting type nat are used for special purposes.

MethodIndex (nat) is used as type for indices into the method component 3.1.2 of the CAP file 3.1 to reference method definition.

ClassIndex (nat) is used as type for indices into the class component 3.1.1 of the CAP file to reference class definition.

StaticIndex (nat) is used as type for indices into the static field component 3.1.3 of the CAP file or the static field image of the machine memory 4.2 to reference a static field.

LocVarIndex (nat) is used as type for indices into the local variables array of the method frame 4.2.2.

FieldIndex (nat) is used as type for indices into the object field array of class instant objects or array objects 4.2.1 or the fields array of the class info 3.1.1.

ExcpIndex (nat) is used as type for indices into the exception array of the method info 3.1.2.

ContextIndex (nat) is used as type for items that denote the context that is assigned to a method 4.2.2 or object 4.2.1.

PCIndex (nat) is used as type for indices into the ByteCodes array of the methodinfo in the CAP file to address an instruction 3.1.2.

The *Lift* type is a parametrized coproduct type with the two components *bot* and *up*.

– TYPE THEORY

$$\begin{aligned}&\mathsf{Lift}\,[\alpha]\ :\ \mathsf{TYPE}\ \stackrel{\text{def}}{=}\\ &\qquad \{\,\mathsf{bot}\colon \mathsf{unit} \mid \mathsf{up}\colon \alpha\,\}\end{aligned} \tag{2.1}$$

The constant *unit* denotes the empty labeled product type []. The type Lift adds a bottom component to a type α and is frequently used to define partial functions. Using the CASE construct we can examine partial functions constructed with Lift and define result terms for for the defined outcome (up) and not defined outcome (bot).

The *RLift* type is simply an extension of the type Lift which adds an additional component *Null* which is of type unit but different to bot.

– TYPE THEORY

$$\begin{aligned}&\mathsf{RLift}\,[\alpha]\ :\ \mathsf{TYPE}\ \stackrel{\text{def}}{=}\\ &\quad\{\,\mathsf{bot}\colon \mathsf{unit} \mid \mathsf{Null}\colon \mathsf{unit} \mid \mathsf{up}\colon \alpha\,\}\end{aligned} \tag{2.2}$$

It can be used to identify results of partial functions with unique properties. In this work it is used to identify the null pointer in a result of type reference. This saves an additional CASE construct that was necessary if the Lift type was used only.

3. CAP File Representation

The CAP file contains all classes and interfaces defined in one Java Card package. Since it is needed to specify the behavior of the Java Card Virtual Machine I will define it first. While the CAP file is initialized and linked on the card, I will not formalize the linking and initialization process. These tasks belong to the Java Card runtime environment. Instead I will assume the CAP file has been linked already and restrict my formalization only to those parts that are needed during the execution process of applets.

In section 3.1 I will formalize the structure of the CAP file needed by my formalization. Section 3.2 explains how the CAP file will be accessed by the Virtual Machine formalization. Some further components, that are needed by the Java Card runtime environment will be discussed briefly in 3.3.

3.1. The CAP File

The JCVM specification [3] describes the CAP file as a stream of 8 bit bytes. In a formalization, however, we are rather interested in the internal structure of the CAP file. The internal structure is defined in a C-like notation using records, which is easy to transform into the type theory and PVS languages.

In the formalization the class structure of the Java Card package is represented the same way as in the CAP file with minor rearrangements to make the formalization simpler. While it is possible to use different methods to represent the class structure we are interested in a complete and accurate representation of the whole JCVM including the CAP file. Also we can formalize the access of the CAP file and the class structure according to how the JCVM accesses it.

The CAP file is constructed of components. Each component describes one aspect of a Java Card package. The components that are needed for defining the semantics of instructions are the class component, method component and static field component. In the JCVM specification [3] the contents of these components is accessed indirectly through the constant pool component, that stores the information necessary to access entities in the other components. However, I choose to access the class, method and static field component directly which makes the constant pool component superfluous. How the constant pool component is eliminated is described in 3.1.4

The *CAP file* is described as a record of a class component, method component and a static field component which are described below.

– TYPE THEORY

$$\mathsf{CAPFile} : \mathsf{TYPE} \stackrel{\text{def}}{=} \big[\mathsf{CC} \colon \mathsf{ClassComponent},\ \mathsf{MC} \colon \mathsf{MethodComponent},\ \mathsf{SC} \colon \mathsf{StaticFieldComponent}\big] \tag{3.1}$$

Smart cards are allowed to contain several applets, since new applets can be loaded onto the card at any time. For technical reasons these applets are stored separately on the card. In order to simplify my formalization I decided to formalize only one CAP file, which contains the class definitions of all applets that are loaded onto the card.

As already stated before, the CAP file is defined as stream of 8 bit bytes. Different types of values in the CAP file are represented by different numbers of bytes. Different instructions are represented by different numbers of bytes too, depending on the number and type of their arguments. This is called a byte oriented approach.

This formalization, however, uses a value oriented approach like [9] and [6]. Value oriented approach means, that one value is represented by exactly one item in the CAP file representation. One instruction in the ByteCodes of the methodinfo for instance is represented and addressed by one instruction-item.

On one hand, value oriented representations are easier to formalize in string and value oriented languages, making it suitable for formalizations using theorem provers. On the other hand byte oriented formalizations like [8] describe the functionality of the virtual machine more precisely than value oriented approaches. It is, for instance, much harder to deal with the byte-size of a stack in value oriented formalization. The size of structures in bytes is therefore not considered in this formalization.

Although this work does not contain a tool that transforms a CAP file into a PVS representation, I'd like to outline which indices and addresses have to be rearranged in the value oriented formalization.

- Offsets into the ByteCodes array of the methodinfo have to be recomputed for jump instructions inside the ByteCodes array and for start offsets in the exceptioninfos.
- Arguments of instructions accessing instance fields must be rearranged. The representation of class instances in the machine memory is described in subsection 4.2.1.
- Arguments of instructions accessing local variables in the method frame have to be recomputed. The representation of method frames in the machine memory is described in subsection 4.2.2.

The only project that is currently working on a JCVM representation including a tool that transforms CAP files is [9]. Unfortunately there is not much information about this project available at this point in time.

3.1.1. Class Component

The class component describes all classes and interfaces contained in a CAP file. The information contained in the class component, as specified in the JCVM Specification, is sufficient to identify a particular class or interface and perform operations on it. These operations include creating a class instance, accessing class instance fields, testing whether or not a cast is valid and dispatching method invocations.

Information necessary for performing verifications is not included in the class component of the JCVM specification, since an implementation is not meant to perform verifications at runtime. Machine formalizations, however, are often used for program verification tasks. Therefore I included additional information in my formalization which normally is stored in the descriptor component. The descriptor component is not part of this formalization but briefly described in subsection 3.1.4.

The class component references other items in the class component and items in the method component. It is referenced by the method component and the static field component.

The *class component* is a list of *Size* classinfos.

– TYPE THEORY

$$\begin{aligned}&\mathsf{ClassComponent} : \mathsf{TYPE} \stackrel{\text{def}}{=}\\&\quad\big[\,\mathsf{Size} \colon \mathsf{ClassIndex},\\&\quad\ \ \mathsf{Classes} \colon \{0 \ldots \mathsf{Size} - 1\} \rightarrow \mathsf{ClassInfo}\big]\end{aligned} \tag{3.2}$$

Each classinfo describes one class or interface, defined in this package. In the JCVM Specification *Size* denotes the number of bytes in the class component. However, in the formalization it makes more sense to specify the number of classinfos. Unlike the JCVM Specification, where items are addressed by byte indices, we address items by their keywords and numbers. Additionally, the range of *Classes* is restricted to 0 ... Size-1 to have a more realistic representation. How the CAP file is read is described in section 3.2.

The *classinfo* is a record describing the look-up information for classes

— TYPE THEORY

$$
\begin{aligned}
&\mathsf{ClassInfo} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
&\quad \big[\, \mathsf{Type}\colon \{\mathsf{Interface}\colon \mathsf{unit} \mid \mathsf{Class}\colon \mathsf{unit}\}, \\
&\quad\ \ \mathsf{SuperClass}\colon \mathsf{ClassIndex}, \\
&\quad\ \ \mathsf{FieldCount}\colon \mathsf{FieldIndex}, \\
&\quad\ \ \mathsf{FieldTable}\colon \{0 \ldots \mathsf{FieldCount} - 1\} \ \to \\
&\qquad\qquad \big[\, \mathsf{Type}\colon \mathsf{MCType}, \mathsf{Token}\colon \{\mathsf{Protected}\colon \mathsf{unit} \mid \mathsf{None}\colon \mathsf{unit}\}\,\big], \\
&\quad\ \ \mathsf{MethodCount}\colon \mathsf{MethodIndex}, \\
&\quad\ \ \mathsf{MethodTable}\colon \{0 \ldots \mathsf{FieldCount} - 1\} \ \to \mathsf{MethodIndex}, \\
&\quad\ \ \mathsf{Context}\colon \mathsf{ContextIndex}\,\big]
\end{aligned}
\tag{3.3}
$$

with fields:

Type indicates if the classinfo describes a class or an interface.

SuperClass points into the class component of this CAP file to the definition of the superclass or super-interface of this class or interface.

FieldCount is the number of fields in this class. FieldCount also includes all fields of all super-classes. The fields of the super-classes have lower indices than the fields defined in this class and the order of fields is the same in all subclasses as in the defining class. This way we can address a field by the same index in all classes it is defined in.

FieldTable assigns every field of the class a type and a field token that defines its modifiers and its type. In the CAP file only *protected* fields are marked with the modifier. *public* and *package* modifiers are not specified in the CAP file, instead they are treated by different instructions. The type MCType is described below.

MethodCount is the number of methods defined for this class. MethodCount also includes all methods of all super-classes for the same reasons as for FieldCount.

MethodTable assigns every method defined in this class an index into the method component to the respective method definition. All information about the methods are stored in the method component which is defined in 3.1.2

Context indicates the package this class is defined in.

The type *MCType* defines a list of types that are supported by the JCVM.

— TYPE THEORY

$$
\begin{aligned}
&\mathsf{MCType} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
&\qquad \big\{\mathsf{Short}\colon \mathsf{unit} \mid \mathsf{Reference}\colon \mathsf{unit}\big\}
\end{aligned}
\tag{3.4}
$$

MCType is used to define the type of items in the CAP file. An extra type is needed because in the JCVM state the types are contained in the memory cell defined in 4.1.1 that is abstracted from the rest or the JCVM and cannot be used in the CAP file.

3.1.2. Method Component

The method component describes all methods defined in a package, including the exception handlers for each method and class initialization methods. The JCVM Specification includes only information into the method component that is needed for method execution. Additional information like modifiers and types are contained in the descriptor component. To have such information available for verification tasks I include necessary data into the method component.

The method component references items in the method component, the class component and the static field component. It is referenced by the class component.

Similar to the class component the *method component* is a list of Size methodinfos.

– TYPE THEORY

$$\begin{aligned}&\mathsf{MethodComponent} : \mathsf{TYPE} \stackrel{\text{def}}{=}\\&\quad\big[\,\mathsf{Size}\colon \mathsf{MethodIndex},\\&\quad\ \ \mathsf{Methods}\colon \{0\ldots \mathsf{Size}-1\} \to \mathsf{MethodInfo}\,\big]\end{aligned} \tag{3.5}$$

Each methodinfo item describes one method. Like in the class component Size specifies the number of methodinfos in the package and the number of methodinfos in the method component is restricted to 0 ... size-1.

The *methodinfo* structure is defined as follows:

– TYPE THEORY

$$\begin{aligned}&\mathsf{MethodInfo} : \mathsf{TYPE} \stackrel{\text{def}}{=}\\&\quad\big[\,\mathsf{Type}\colon \big\{\mathsf{Clinit}\colon \mathsf{unit} \mid \mathsf{Virtual}\colon \{\mathsf{Protected}\colon \mathsf{unit} \mid \mathsf{None}\colon \mathsf{unit}\}\big\},\\&\quad\ \ \mathsf{Class}\colon \mathsf{ClassIndex},\\&\quad\ \ \mathsf{nArgs}\colon \mathsf{nat},\\&\quad\ \ \mathsf{ArgType}\colon \{0\ldots \mathsf{nArgs}-1\} \to \mathsf{MCType},\\&\quad\ \ \mathsf{maxPC}\colon \mathsf{PCIndex},\\&\quad\ \ \mathsf{ByteCodes}\colon \{0\ldots \mathsf{maxPC}-1\} \to \mathsf{Instruction},\\&\quad\ \ \mathsf{ExceptionCount}\colon \mathsf{ExceptionIndex},\\&\quad\ \ \mathsf{Exceptions}\colon \{0\ldots \mathsf{ExceptionCount}-1\} \to \mathsf{ExceptionInfo},\\&\quad\ \ \mathsf{Context}\colon \mathsf{ContextIndex}\,\big]\end{aligned} \tag{3.6}$$

with fields:

Type specifies the type of the method. It can be either a virtual method (virtual) or a class initialization method (Clinit). If the method is of type virtual, then the type contains additional modifier information.

Class is an index into the class component pointing to the class that first defined this method. It does not point to the class that defines exactly this instance by overwriting the method.

nArgs specifies the number of arguments for this method plus one.

ArgType assigns each argument of this method a type.

maxPC is the number of instructions in the ByteCodes array of this method.

ByteCodes is an array of size maxPC which maps a bytecode index to an instruction and its arguments. The JCVM specification defines the ByteCodes array as a stream of bytes where each instruction consists of one byte for the instruction opcode and several bytes for the arguments of the instruction. Thus in the CAP file specification only indices that point to an instruction opcode are valid indices. Indices to instruction arguments are not allowed. Due to the value oriented approach of this formalization, however, there is exactly one instruction defined for each index into the ByteCodes array that is smaller than maxPC.

ExceptionCount specifies the number of throwable exceptions of this class. The exceptions that may be thrown by a method are specified in the Java source code with the *throws* statement.

Exceptions gives a list of exceptioninfos for these methods. The exceptioninfos describe the type, active range and the corresponding handler for each exception that can be thrown by this method. The Exceptions array is used to identify the handler of an exception that is thrown inside this method.

Context specifies the package this method was defined in. It can be different to the package of the defining class indicated by Class since methods can be overwritten in inherited classes.

The *Instructions* in the ByteCodes array are defined by a coproduct containing a component for each instruction known by the machine.

– TYPE THEORY

$$\begin{aligned}&\mathsf{Instruction} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=}\\ &\quad\{\, \mathsf{new}\colon [\mathit{class}\colon \mathsf{ClassIndex}\,],\\ &\quad\;|\; \mathsf{invokevirtual}\colon [\mathit{method}\colon \mathsf{MethodIndex}, n : \mathsf{nat}\,],\\ &\quad\;|\; \mathsf{return}\colon \mathsf{unit}\\ &\qquad\vdots\\ &\quad\}\end{aligned} \tag{3.7}$$

The *exceptioninfo* contains all information to uniquely identify the exception handler that corresponds to an exception that can be thrown by this method. Exception handlers are catch clauses and finalize clauses which are defined inside the ByteCodes array of the respective method.

– TYPE THEORY

$$\begin{aligned}&\mathsf{ExceptionInfo} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=}\\ &\quad[\,\mathsf{Start}\colon \mathsf{PCIndex},\\ &\quad\;\mathsf{Length}\colon \mathsf{nat},\\ &\quad\;\mathsf{Handler}\colon \mathsf{PCIndex},\\ &\quad\;\mathsf{Type}\colon \mathsf{ClassIndex}\,]\end{aligned} \tag{3.8}$$

The fields in the exceptioninfo are:

Start is an index into the ByteCodes array of this method indicating the start of the active range. The active range is the range in the ByteCodes array in which the exception may be caught.

Length indicates the length of the active range of this exceptioninfo. The active range is defined including Start+Length.

Handler is the index into the ByteCodes array of this method to the first instruction of the exception handler.

Type indicates the type of the exception of this exception handler. If the exception is a user defined exception, then Type indices the definition of the exception class in the Classes field of the class component. If the exception is predefined by the machine then Type is the predefined value for the machine defined exception.

In the JCVM specification the exceptioninfos are not defined in the methodinfo but in the method component structure. The start index is simply a byte offset into the

ArithmeticException
ArrayStoreException
ClassCastExeption
IndexOutOfBoundException
ArrayIndexOutOfBoundException
NullPointerException
SecurityException
NegativeArraySizeException

Figure 3.1.: Predefined exception of the JCVM

method component. In a value oriented formalization, however, it is not possible to have indices point to an arbitrary position in a structure. Therefore I chose to define the exceptioninfos in the methodinfo they are associated with to know which method the exceptioninfo is assigned to. It is of course possible to uniquely assign an exception handler to a method, since neither exception handlers nor ByteCodes of methods are allowed to overlap.

According to the JVM specification the exceptioninfos in the exception field of the methodinfo are sorted in a way, such that the first matching handler is the one that handles the exception.

The predefined exceptions of the JCVM are listed in figure 3.1.

While all of the predefined exceptions are currently supported by the formalization there exists no class definition for throwable or for these specific exceptions. The predefined exceptions are simply addressed by their predefined values of type nat. Errors are not supported by this formalization since no special behavior for errors is defined in the JCVM specification except that the machine halts.

The exact mechanism of throwing and handling of exceptions is described and formalized in section 5.4.

3.1.3. Static Field Component

The static field component, as defined in the JCVM specification, holds all information for creating and initializing the static field image including static arrays. Since this formalization does not cover applet initialization I only use a very simplified static field component that only holds information needed for verifications.

The static field component references elements of the class component. The method component references the static field component.

Like the other components the *static field component* is a list with a fixed size containing static field infos.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{StaticFieldComponent} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
&\qquad \big[\,\mathsf{Size}\colon \mathsf{StaticIndex}, \\
&\qquad\ \ \mathsf{StaticFields}\colon \{0 \ldots \mathsf{Size}-1\} \to \ [\,\mathsf{Modifier}\colon \{\mathsf{Protected}\colon \mathsf{unit} \,|\, \mathsf{None}\colon \mathsf{unit}\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Class}\colon \mathsf{ClassIndex}\,]\,\big]
\end{aligned} \tag{3.9}
$$

Since the static field info contains only 2 fields it is not defined in an extra structure. Please note that this definition differs strongly from the JCVM specification where it mainly contains data for the initialization of classes and the static field image. Final static fields are not represented in the static field component since they are treated as constants in the bytecode.

Each element of *StaticFields* describes exactly one static field. The Modifier item describes the modifiers of this static field. The Class item is a pointer into the class component to the class this static field belongs to. In this formalization the Class item is mainly used to identify the package in which the static field was defined.

3.1.4. CAP file Formalization Issues

The CAP file as described in the JCVM specification is much more compact than a Java Class file, generated by a Java compiler. Unlike the Java Class file that is also used for bytecode verifications, the CAP files main purpose is to execute applets on very resource restricted devices like smart cards. Thus, lots of information that is contained in the Java class file has been omitted in the CAP file. Additionally, the data in the CAP file has been heavily restructured.

Despite these CAP file optimizations, some tasks of the JCVM (e.g. method and instance field dispatching that are of minor importance for verifications) are still unavoidably complex and time consuming. A formalization, however, is mainly used for verification tasks instead of the execution of programs. In order to keep the formalization simple enough to fit into the frame of this work and to focus on important aspects, I had to simplify my formalization of the CAP file.

Some issues like omitting the descriptor component are already mentioned above. The elimination of the constant pool, the addressing of class members and the treatment of modifiers in the CAP file will be discussed in this subsection.

Constant Pool Elimination

In the CAP file the constant pool is a central look-up register. Classes and class members are not addressed directly inside the CAP file. Instead all classes and class members are accessed through entries in the constant pool component making it a central part of the CAP file. The constant pool entries contain sufficient information to uniquely identify and address items in the CAP file and machine memory.

The constant pool component is especially important for the process of applet linking. Classes and class members, that are imported into a package have an entry in the constant pool as well as those defined in the package. These constant pool entries are resolved during the linking such that the respective items can be accessed from inside the package.

In order to simplify the formalization I decided to address items in the CAP file directly like it is described in [9]. This is possible, since this formalization does not deal with applet linking and all CAP files are combined in one single structure. Other formalizations like [6] and [8] include the constant pool.

Addressing items directly has the following advantages:

- The structure of the CAP file becomes smaller.
- The CAP file representation is much easier to read, since items are addressed directly. This is a big advantage because there is currently no tool for translating CAP files into a PVS representation and the PVS representation therefore must be written by hand.
- Constant pool entries don't have to be resolved anymore. This simplifies the CAP file access and lets us concentrate on important issues.

However, I recommend someone who formalizes a Java Card Virtual Machine to consider seriously whether to include the constant pool or not, depending on the purpose of the formalization. It turned out to be quite tricky to achieve without all the data contained in the constant pool.

Identifying Class Members

The JCVM specification states that the method table of the classinfo only has to include methods defined by this class and can include some or all methods defined in super-classes of this class. The same is stated for the field table of the classinfo. It might be necessary to propagate method invocations through the class hierarchy to find the right method or class field.

Class member allocations are usually not subject to verifications. To avoid complex class member allocation mechanisms I decided to include all class members of this class and all super-classes in the method table and field table respectively. The class members are sorted by their appearance in the class hierarchy such that those defined in super-classes come first.

This way the same method can be accessed with the same index throughout all classes it is accessible from, whether it is overwritten by this class or not.

Treating Modifiers

While the Java Card language supports all modifiers that are supported by Java, most modifiers of classes and class members are not stored in the CAP file. The only modifier that is actually checked during run time is the protected modifier. That's why protected is the only modifier included in the formalization.

The other modifiers are treated differently and resolved at compile time. Final static fields for instance are not contained in the CAP file at all, but are included as constants in the bytecode. Thus it seems like modifiers are ignored by the JCVM. But this is wrong.

Java Card uses two mechanisms to enforce modifiers.

- Private and package visible items in the CAP file are simply not contained in the export file. Since they are not in the export file, they cannot be linked to other packages. These items are not visible and thus not accessible from other packages.
- Java Card uses different instructions for some modifiers. Private method calls, for instance, are realized by a different instruction than public method calls.

The JCVM therefore depends on the compiler to treat modifiers correctly. Errors concerning modifiers can generally not be detected by the JCVM. Also there is no bytecode verifier specified for CAP files like for Java class files.

3.2. Reading the CAP file

In order to process the instructions in the bytecode and to verify the execution, the JCVM has to read data from the CAP file. This data is instructions as well as class and method properties. The way I defined the structure would allow us to access items in the CAP file directly. Many items like classinfos, methodinfos and staticfieldinfos, however, are defined in a certain range that is indicated by a certain size item.

PVS forces us to verify if the accessed item is inside the defined range. If we don't verify if an item is defined ourself then PVS generates type check constrains that have to be proven separately. Such proofs are in general not feasible. We can for instance not prove, if a class definition, that we want to read data from, actually exists. Furthermore, such properties are good candidates for verifications.

Therefore we have to check range constraints before we can read data. A term, that represents the superclass of a class x in the CAP file p would look like this:

– TYPE THEORY

$$\mathsf{IF}\ x < p.\mathsf{ClassComponent.Size}\ \mathsf{THEN}\ p.\mathsf{ClassComponent.Classes}(x).\mathsf{SuperClass}$$
$$\mathsf{ELSE}\ \ldots$$

The definition of the operational semantics, defined in chapter 5, does usually not access the CAP file directly. Class properties are either read by the memory cell abstraction functions defined in section 4.1.1 or by special functions designed for CAP file access. This way the structure of the CAP file is mostly hidden from the operational semantics that makes it easier to change the representation of the CAP file.

The functions that access the CAP file take a certain number of arguments, that indicate the items to be read, and returns either a value of type Lift[type], in case an item shall be read, or type boolean if a property of the class definition is checked.

The following example demonstrates an access function of type boolean. The predicate *isSubClass* checks if *sub* is a subclass of *sup* in the class component *cc*. To achieve this it propagates upwards through the class hierarchy of *sub* until it finds *sup* or no superclass is defined anymore.

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{isSubClass}\colon \mathsf{ClassIndex} \times \mathsf{ClassIndex} \times \mathsf{ClassComponent} \to \mathsf{boolean} \\
\quad \stackrel{\text{def}}{=}\ \lambda sub\colon \mathsf{ClassIndex}, sup\colon \mathsf{ClassIndex}, cc\colon \mathsf{ClassComponent}. \\
\qquad\quad \mathsf{IF}\ sub = \mathsf{noSuperClass} \\
\qquad\qquad\quad \mathsf{THEN\ false} \\
\qquad\quad \mathsf{ELSE\ IF}\ sub = sup \\
\qquad\qquad\quad \mathsf{THEN\ true} \\
\qquad\quad \mathsf{ELSE\ IF}\ sub < cc.\mathsf{Size} \\
\qquad\qquad\quad \mathsf{THEN\ isSubClass}(cc.\mathsf{Classes}(sub).\mathsf{SuperClass}, sup, cc) \\
\qquad\qquad\quad \mathsf{ELSE\ false}
\end{array}
$$

Please note, that this function is defined recursive. It does not yet check for recursive class inheritance. The constant *noSuperClass* is used to indicate that a class has no superclass.

The second example demonstrates a function of type Lift. The function *GetInstruction* returns the instruction addressed by program counter *pc* in method *m* of method component *mc*.

— TYPE THEORY

$$\begin{aligned}
&\mathsf{GetInstruction}\colon \mathsf{MethodIndex} \times \mathsf{PCIndex} \times \mathsf{MethodComponent} \to \mathsf{Lift}[\mathsf{Instruction}]\\
&\quad \stackrel{\text{def}}{=} \lambda m\colon \mathsf{MethodIndex}, pc\colon \mathsf{PCIndex}, mc\colon \mathsf{MethodComponent}.\\
&\qquad \mathsf{IF}\ m < mc.\mathsf{Size} \land pc < mc.\mathsf{Methods}(m).\mathsf{maxPC}\\
&\qquad\quad \mathsf{THEN}\ \mathsf{up}\, mc.\mathsf{Methods}(m).\mathsf{ByteCodes}(pc)\\
&\qquad \mathsf{ELSE}\ \mathsf{bot}\ \mathsf{ENDIF}
\end{aligned}$$

bot indicates that an error has occurred and the instruction can not be returned. If the instruction could be found, then the function returns up containing the Instruction.

3.3. Further Components

As already mentioned, there are some more components in the CAP file. These components were not included in the formalization, since they are not used by the JCVM. Some of them, however, are crucial for the JCRE especially import and applet component. At least the following CAP file components should be mentioned for modification or inclusion into a JCRE formalization:

Export component The export component lists all static elements that can be imported by classes in other packages. Instance fields and virtual methods are not represented in the export component, but in the export file associated with this package.

Import component The import component contains the list of packages imported by classes in this package. It is needed for package linking.

Applet component The applet component lists all applets defined in a package. It is especially important since there can be more than one applet defined in one package. For each applet it specifies an applet name and an index into the method component pointing to the applet installation method. Applet initialization is not formalized in this work.

Static field component The static field component in this formalization is very simplified. A complete static field component would also contain all information necessary to create and initialize a static field image, defined as static memory in the memory model in section 4.2.

Descriptor component The descriptor component contains additional information about elements of the class component, method component and static field component. It is only needed to parse and verify this CAP file. In this formalization all needed information from the descriptor component is included into the respective elements of other components.

4. The Machine State

In this formalization the JCVM memory is represented as a state, making this a state-based formalization. While the memory of an actual JCVM is an array of bytes we are rather interested in the way the memory is structured. The structure of the JCVM state is formalized as memory model in section 4.2.

The representation and access of primitive and reference types is hidden from the memory model and the operational semantics. This abstraction, which will be described in section 4.1, makes the formalization more flexible. So we can easily describe defensive implementations of the JCVM as well as non-defensive ones without modifying the memory model or the operational semantics.

How the memory is read is described in section 4.3. How it is updated is described as part of the operational semantics in section 5.3.

4.1. Modeling Primitive Types and Reference Types

The JCVM distinguishes 2 kinds of data-types: primitive types and reference types. The values that are stored in these data types are referred to as *primitive values* and *reference values* respectively.

The support for primitive data types is restricted to the cardinal types *byte*, *short* and optionally *int*. Floating point numbers and characters are not supported by the JCVM since they are not needed in a typical Java Card environment. The advantage of omitting floating point types and characters is that the JCVM does not need instructions to deal with these types and thus has been optimized in size and complexity.

The *reference type* is used to store references into the heap memory. Objects referenced by reference values are class instances and arrays. The support for reference types is identical to the JVM.

In this formalization I will only consider data type short as example for primitive data-types and type reference as reference data-type. The formalizations of types int and byte would only differ in their defined range from the type short and can be added easily whenever they are needed.

4.1.1. The Memory Cell Abstraction

The JCVM specification doesn't dictate any particular way to store data-type values. It only ensures that sufficient information about a particular value can be stored. For that reason it defines an abstract storage unit *word* and mandates, that one word must be large enough to hold one value of type *byte*, *short* or *reference* and two *words* must be large enough to hold a value of type *int*.

The actual size of a *word* and how much additional information about the value is stored is not given by the JCVM. Thus the implementor of a JCVM has all the freedom to decide how large a *word* is and can optimize his implementation to its very purpose. It is for instance possible to store type information in a word and perform exhaustive type checking during execution if the application field requires runtime type-checks. If execution speed is more important, then one might rather want to implement a non-defensive JCVM without type checking.

In order to cover all possible implementations of a JCVM I decided to abstract the storage of data-type values in the type *MemCell*.

– TYPE THEORY

$$\mathsf{MemCell}\colon \mathsf{TYPE} \tag{4.1}$$

The memory cell *MemCell* can store a value of any data-type including references. The actual implementation of MemCell is hidden from the memory model and the operational semantics.

In PVS this abstraction is realized by giving the memory cell formalization as a parameter to those theories that depend on MemCell. This mechanism makes it especially easy to switch between specifications since only parameters of imported theories have to be changed. A similar way to abstract the specification of the storing unit is described by Push [6]. It does, however, not explain how a different specification is set up and also does not provide such an alternative specification.

Now, if the formalization of the memory cell is hidden from the remaining formalization, we also have to specify functions for accessing and modifying memory cells. A prototype for the type of these memory cell operations is outlined in the structure *MemCellOperations*.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{MemCellOperations[MemCell]} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
&\quad \big[\,\mathsf{Null}\colon \mathsf{MemCell}, \\
&\quad \ \ \mathsf{NewReference}\colon \mathsf{MemCell} \to \mathsf{MemCell}, \\
&\quad \ \ \mathsf{isShort}\colon \mathsf{MemCell} \to \mathsf{boolean}, \\
&\quad \ \ \mathsf{toShort}\colon \mathsf{int} \to \mathsf{MemCell}, \\
&\quad \ \ \mathsf{initMemCell}\colon \mathsf{MCType} \to \mathsf{MemCell}, \\
&\quad \ \ \mathsf{getReference}\colon \mathsf{MemCell} \to \mathsf{RLift[MemCell]}\,\big]
\end{aligned}
\tag{4.2}
$$

An exact implementation of MemCellOperations must be given for every memory definition that shall be used in a formalization. The MemCellOperations are then given as parameter to the respective theories as well.

The items in MemCellOperations have the following meaning:

Null defines a representation of the *null pointer*. The null pointer is a reserved value of type reference that is used for references that do not point to any object on the heap.

NewReference computes a reference pointing to a memory position that is empty and can store a newly created object. The parameter of type MemCell has to be the topmost used position on the heap. Since the JCVM does not perform garbage collection it is assumed here that the heap is filled successively like a stack.

isShort is used to verify if a memory cell is of type short. If the memory cell is of type short then it returns true otherwise false.

toShort returns a memory cell of type short which represents a given value.

initMemCell is used to define a standard value to which a memory cell of a certain type is initialized. This function is used to initialize instance fields. It can also be used to initialize static fields for which no initial value is defined in the CAP file. Static field image initialization, however, is not covered by this formalization.

getReference is used to verify if a given memory cell is if type reference. If the memory cell is not a reference then it returns bot. If the memory cell represents the null pointer, then it returns Null otherwise it returns Up and the given memory cell.

The defined MemCellOperations only outlines what items are needed in a complete formalization. The actual formalization contains functions for accessing and manipulating all data-types in all places the MemCell-type is used.

The way MemCellOperations is formalized gives the opportunity to contain inconsistencies. The function getReference may return Null for a memory cell different from the specified null pointer. Such inconsistencies are in most cases not welcomed in a formalization. In order to deal with them I introduce a predicate *MemCellConstraints* which states properties that must hold for consistent MemCellOperations. For the memory cell operations example above, the following memory cell constraints may be given

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{MemCellConstraints\colon MemCellOperations} \to \mathsf{boolean}\\
&\quad \stackrel{\text{def}}{=} \lambda \mathsf{MCO\colon MemCellOperations.}\\
&\qquad \mathsf{MCO.isShort(MCO.initMemCell(Short))} \land\\
&\qquad \mathsf{MCO.getReference(MCO.initMemCell(Reference))} \neq \mathsf{bot} \land\\
&\qquad \forall x\colon \mathsf{int.\ MCO.isShort(MCO.toShort}(x)) \land\\
&\qquad \forall x\colon \mathsf{MemCell.\ MCO.getReference(MCO.NewReference}(x)) \neq \mathsf{bot} \land\\
&\qquad \forall x\colon \mathsf{MemCell.\ MCO.NewReference}(x) \neq \mathsf{MCO.Null} \land\\
&\qquad \forall x\colon \mathsf{MemCell.\ MCO.getReference}(x) = \mathsf{Null} \iff x = \mathsf{MCO.Null}
\end{aligned}
$$

stating that the initialized memory cells must be recognized as of the respective types, isShort must be true if applied on a memory cell generated with toShort, a reference generated by the function NewReference must be recognized as of type reference and not be the value reserved for the null pointer and getReference only returns Null for the null pointer.

These constraints may be proven for a particular set of memory cell operation. In the PVS formalization, the theories that use the MemCellOperations import them as a subtype of the MemCellOperations-type for which the constraints hold. By using this PVS feature we assume that the constraints hold automatically. PVS takes over the task to generate type-check constraints for the actually imported MemCellOperations. These type-check constrains can then easily be proven with the PVS prover.

This memory cell abstraction will allow the formalization of the operational semantics exactly as it is given in the JCVM specification while preserving the flexibility given to implementers of the JCVM. Issues that can be decided individually can all be formalized in the MemCellOperations. Thus it should not be necessary to change the operational semantics or the memory model in order to adapt the formalization to some behavior inside the admissible boundaries.

In the next subsections I will give two examples of memory cell definitions, one for a strongly typed memory cell and one for a memory cell without type information.

4.1.2. An Untyped Memory Cell

The *untyped memory cell* formalized as *UTMC* does not store any type information about the values it stores. All references and cardinals are stored as an integer value.

– TYPE THEORY

$$\mathsf{UTMC}\colon \mathsf{MemCell} \stackrel{\text{def}}{=} \mathsf{int} \tag{4.3}$$

This example is intended to resemble the behavior of an actually implemented JCVM. Instead of storing type information and performing type checks, such a JCVM would rely on the compiler that generated the class files. Since people who load applets onto a smart cards usually know the origin of the applet, it doesn't cause any confidentiality problems. Unfortunately there is no bytecode verifier defined for Java Card applets like it is for Java class files. So we have to trust the compiler to perform source code verifications and generate well formed CAP files.

This formalization also discards all other verifications that could be done by the machine. It therefore resembles the most non-defensive implementation of a JCVM possible. That makes the formalization of MemCellOperations as *UTMCOps* very simple.

– TYPE THEORY

$$\begin{aligned}
&\mathsf{UTMCOps}\colon \mathsf{MemCellOperations}[\mathsf{UTMC}]\\
&\quad \stackrel{\text{def}}{=} \;\big(\mathsf{Null} = 0,\\
&\qquad\quad \mathsf{NewReference} = \lambda x\colon \mathsf{UTMC}.\, x+1,\\
&\qquad\quad \mathsf{isShort} = \lambda x\colon \mathsf{UTMC}.\, \mathsf{true},\\
&\qquad\quad \mathsf{toShort} = \lambda x\colon \mathsf{int}.\, x,\\
&\qquad\quad \mathsf{initMemCell} = \lambda x\colon \mathsf{MCType}.\, 0,\\
&\qquad\quad \mathsf{getReference} = \lambda x\colon \mathsf{UTMC}.\, \mathsf{IF}\ x = 0\ \mathsf{THEN}\ \mathsf{Null}\ \mathsf{ELSE}\ \mathsf{up}\, x\big)
\end{aligned}$$

Instead of defining a special representation for the null pointer, the cardinal number zero is used which also represents the short value 0. New references are computed by simply increasing the value of the given reference. All other items in the MemCellOperations return either true or up respectively.

4.1.3. A Typed Memory Cell

A *typed memory cell* is defined as a coproduct that reserves a representation for every data-type known by the machine. It is formalized as *TMC*.

– TYPE THEORY

$$\begin{aligned}
\mathsf{TMC} : \mathsf{TYPE} \stackrel{\text{def}}{=}& \\
&\{\mathsf{Short}: \mathsf{int} \\
&\mid \mathsf{Reference}: \mathsf{Lift}[\mathsf{nat}]\}
\end{aligned} \tag{4.4}$$

Since this formalization only considers types short and reference, there is only a representation for these two types. The cardinal type short is simply defined as integers. The range of the data-types is not yet considered by this formalization. The reference type is defined as Lift, making it possible to distinguish the null pointer from other references.

The purpose of this defensive formalization is to store full type information and perform all possible runtime checks. Program errors and not well formed CAP files would be detected by this formalization and affect the computation by returning the result Undef (see section 5.2). This formalization would typically be used for verification tasks. A JCVM that is used in very critical environments which require applet execution to be specially safe may be implemented after this formalization too.

The MemCellOperations for the typed memory cell could be defined as follows:

– TYPE THEORY

$$\begin{aligned}
&\mathsf{TMCOps}: \mathsf{MemCellOperations}[\mathsf{TMC}] \\
&\stackrel{\text{def}}{=} \big(\mathsf{Null} = \mathsf{Reference\,bot}, \\
&\quad \mathsf{NewReference} = \lambda x: \mathsf{TMC}.\ \mathsf{CASES}\ x\ \mathsf{OF}\ \{ \\
&\qquad \mid \mathsf{Short}\ s \mapsto \mathsf{up}\,0 \\
&\qquad \mid \mathsf{Reference}\ r \mapsto\ \mathsf{CASES}\ r\ \mathsf{OF}\ \{ \\
&\qquad\qquad \mid \mathsf{bot} \mapsto \mathsf{up}\,0 \\
&\qquad\qquad \mid \mathsf{up}\,n \mapsto \mathsf{up}\,(n+1)\,\}\,\}, \\
&\quad \mathsf{isShort} = \lambda x: \mathsf{TMC}.\,\mathsf{CASES}\ x\ \mathsf{OF}\ \{\ \mid \mathsf{Short}\ s \mapsto \mathsf{true}\ \mid \mathsf{Reference}\ r \mapsto \mathsf{false}\,\}, \\
&\quad \mathsf{toShort} = \lambda x: \mathsf{int}.\,\mathsf{Short}\ x, \\
&\quad \mathsf{initMemCell} = \lambda x: \mathsf{MCType}.\,\mathsf{CASES}\ x\ \mathsf{OF}\ \{\ \mid \mathsf{Short} \mapsto \mathsf{Short}\,0 \\
&\qquad \mid \mathsf{Reference} \mapsto \mathsf{Reference\,bot}\,\}, \\
&\quad \mathsf{getReference} = \lambda x: \mathsf{TMC}.\ \mathsf{CASES}\ x\ \mathsf{OF}\ \{ \\
&\qquad \mid \mathsf{Short}\ s \mapsto \mathsf{bot} \\
&\qquad \mid \mathsf{Reference}\ r \mapsto\ \mathsf{IF}\ r = \mathsf{bot}\ \mathsf{THEN}\ \mathsf{Null}\ \mathsf{ELSE}\ \mathsf{up}\,x\,\}\big)
\end{aligned}$$

with the single items having the following meaning:

Null The reference type is defined as a Lift-type. So the bot can be used to represent the null pointer. This way the null pointer and actual references are clearly separated and thus can be easily distinguished.

NewReference In this formalization I choose to represent object references by successive natural numbers. A new reference thus is computed by increasing the topmost reference by one. The topmost reference is stored in the JCVM state which is defined in section 4.2. In my formalization it is not possible to give the whole state as argument to MemCellOperations since the definition of the state depends on the type MemCell. So the topmost reference from the JCVM state must be given as argument to NewReference and the topmost reference must be updated afterwards to the newly computed reference. In case the argument of NewReference is not a valid reference Reference 0 is returned which represents the first reference.

isShort In contrast to the untyped memory cell this version verifies if the argument is of type short and returns the respective result.

toShort returns a memory cell of type short which contains the number given as argument.

initMemCell defines Short 0 as initial value for type short and Reference bot as initial value for type reference. The fact that the initial value for type reference is equal to the null pointer is a coincidence that doesn't mean anything.

getReference In addition to the untyped memory cell this version verifies if the argument is really a reference. If it is not a reference, then getReference returns bot. If the argument is a reference then getReference behaves like the untyped version.

4.2. The Memory Model

In this section I will define a model of the JCVM state. The state represents the memory which holds all runtime data of the JCVM during execution of applets. The memory model of this formalization will be restricted to those parts that are needed by the JCVM. A formalization of the JCRE will have to store additional information.

While the memory cell, defined in the previous section, holds actual data values, the memory model defines how these data values are arranged in the machine memory. The memory model defines a structures consisting of all entities contained in the the JCVM memory. These entities are methods that are invoked during execution of an applet and objects.

Unlike the memory of an actual machine, which is an array of bytes, the memory of this formalization is represented by a record structure like the CAP file. This makes it easier to implement a representation of the structure in theorem provers but makes it difficult to deal with the actual size of structures. Since Java Card is designed for use in resource restricted devices, the size of structures would be an interesting issue for verifications. This formalization, however, will not deal with the size of structures.

The structure of the memory is analogous to the one in [10] described by the record *State*.

– TYPE THEORY

$$\begin{aligned} &\mathsf{State} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ &\quad \big[\, \mathsf{HeapTop} \colon \mathsf{MemCell}, \\ &\quad\; \mathsf{HeapMem} \colon \mathsf{MemCell} \to \mathsf{ObjectCell}, \\ &\quad\; \mathsf{StackMem} \colon \mathsf{list}[\mathsf{FrameCell}], \\ &\quad\; \mathsf{StaticMem} \colon \mathsf{StaticIndex} \to \mathsf{MemCell} \,\big] \end{aligned} \tag{4.5}$$

HeapTop is the reference that addresses the next free position on the heap memory. A new object is always created at this address. When a new object was created the new HeapTop must be computed by the NewReference functions from the respective MemCellOperations with the old HeapTop as argument.

HeapMem The heap memory is an array that stores all objects that are created during execution of an applet. Objects are addressed by memory cells of type reference. A single object is represented by the ObjectCell structure. The heap memory is discussed in detail in subsection 4.2.1.

StackMem The stack memory is responsible for storing execution data of methods. It is represented by a stack of frames. Each frame represents one method. The stack memory is discussed in detail in 4.2.2.

StaticMem The static field memory stores the static fields of all classes of the CAP file. Since there is only one instance for each static field of a class, the static fields are stored separately from the class instances. Each static field is represented by a memory cell which is addressed by the same index as its representation in the static field component of the CAP file. Static arrays are not stored in the static field memory. Instead they are represented by array objects which are created by the respective class initialization methods. The image of the static field memory is created and initialized during the applet initialization which is not included in this formalization.

Other formalizations like [6] and [9] include exception flags and more information about the result of computations into the machine state. In this formalization the state will be restricted to data needed during computations. Computation results will be represented by the Result-type which is discussed in 5.2. This is typical for a coalgebraic approach.

The memory layout of the JCVM is similar to the memory layout of most object oriented programming languages. So it's no surprise that other formalizations have similar memory layout. Basically they only differ in the layout of the static fields memory that may be organized in classes rather than a list of single static fields. Most formalizations, however, do not include the static field memory.

A memory model for the Java language is described in [10]. Unlike in virtual machines, it organizes the stack memory as an array instead of a stack. The Java semantics requires any stack entry to be accessible at any time, since it is not defined as sequential execution of instructions like the virtual machine semantics. For the JCVM semantics (formalized in chapter 5) only the topmost stack entry is relevant for computations.

A JCRE additionally would need a context stack where information about the context hierarchy is stored. The use of contexts is briefly described in section 5.3.

4.2.1. Heap Memory

The heap memory is the place where the virtual machine stores data objects that are created during execution. There are two kinds of objects used by the JCVM, class instances and arrays. The type of an object is specified by the structure *ObjectType*.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{ObjectType} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
&\quad \{\, \mathsf{Array} \colon [\, \mathsf{Type} \colon \mathsf{ArrayType}, \mathsf{Count} \colon \mathsf{FieldIndex} \,] \\
&\quad \mid \mathsf{Class} \colon \mathsf{ClassIndex} \,\}
\end{aligned}
\tag{4.6}
$$

The coproduct contains the component Class for class instances and Array for arrays. The type definition of the class instance object stores an index into the CAP file pointing to the definition of the class. This index is used to read class information about the class instance that is used during execution. The class instance object only stores runtime data in the object cell that is defined below.

The type definition of the array object stores the number of fields in the array, labeled with Count, and the type of the fields in the array, labeled with Type. The type *ArrayType* is defined as a coproduct containing an element for all possible array field types namely short and boolean in this formalization.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{ArrayType} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
&\quad \{\, \mathsf{Short} \colon \mathsf{unit} \mid \mathsf{Boolean} \colon \mathsf{unit} \,\}
\end{aligned}
\tag{4.7}
$$

Arrays of type reference are not represented as array objects. Instead these arrays are stored as class instances and have a definition in the CAP file. The type boolean only exists in arrays. When boolean array fields are read from an array they are converted into an other type that may be represented by the MemCell-type.

The objects are represented by the *ObjectCell*-structure.

– TYPE THEORY

$$\begin{aligned} &\mathsf{ObjectCell}\,[\mathsf{MemCell}]\ :\ \mathsf{TYPE}\ \stackrel{\text{def}}{=} \\ &\qquad \big[\,\mathsf{Type}: \mathsf{ObjectType}, \\ &\qquad\ \ \mathsf{Field}: \mathsf{FieldIndex} \rightarrow \mathsf{MemCell}, \\ &\qquad\ \ \mathsf{Context}: \mathsf{ContextIndex}\,\big] \end{aligned} \tag{4.8}$$

The JCVM specification does not mandate any particular layout of the object storage and the heap memory in general. So the object cell only represents a record with fields for all data that must be stored in an object. The fields in this record have the following meaning:

Type indicates the type of the object as described above.

Field is the list of instance fields or array fields of this object represented by an array of (parameter) memory cells. The fields are addressed by indices starting with 0.

Context Every object is owned by a context. The owning context is determined when the object is created and is identical to the context of the method it was created by. The context an object is owned by is used to enforce security mechanisms.

The object cells are addressed by memory cells of type reference and stored successively in an array starting with the lowest array index. The array fields of the heap memory are initialized to a special object cell constant. The size of the heap memory is not restricted and any reference can be used to address an object whether an object was created for that reference or not. Object creation and initialization is described as part of the operational semantics in subsection 5.3. Like in a real memory it is possible to read from or write into non-existing objects with undefined results. If an object field is defined must be verified by the operational semantics by fetching the respective information from the object type or class definition.

The JCVM specification does not define garbage collection for the heap memory. Thus, all created objects are preserved until the JCVM terminates, even if no reference to an object exists anymore. Also it is not possible for an applet to erase objects from the heap.

The JCRE may also create objects in the heap memory that belong to the JCRE. Such objects are only used by JCRE functions and not by the JCVM, so they are not included in this formalization.

4.2.2. Stack Memory

The stack memory is the place where the runtime data about the methods is stored. This includes information about the methods as well as the data that is processed during the execution of the respective methods. Each method that is executed is represented by a frame. Since only one method is executed at a time, only the frame that belongs to the currently executed method is relevant for computations. This method and frame are called current method and current frame respectively.

A new frame is created whenever a method is invoked. It is removed when the execution of the method terminates due to a return instruction or when no exception handler for a thrown exception could be found in this method. Unlike objects, frames only exist during the lifetime of a method. After a frame has been removed, there will no more data be available about the corresponding method execution. These properties are exactly described by a stack. That's why all virtual machine formalizations do so.

The frame whose method is executed is always the topmost frame on the stack. A newly created frame is pushed onto the stack making it the current frame. When a method execution terminates, the topmost frame is popped from the stack. The next topmost frame that belongs to the method, that called the terminated method, becomes current. The execution continues in that method.

Method frames are represented by the *FrameCell*-structure.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{FrameCell[MemCell]} : \mathsf{TYPE} \stackrel{\text{def}}{=}\\
&\quad \big[\, \mathsf{Method} \colon \mathsf{MethodIndex},\\
&\quad\ \ \mathsf{PC} \colon \mathsf{PCIndex},\\
&\quad\ \ \mathsf{Class} \colon \mathsf{ClassIndex},\\
&\quad\ \ \mathsf{LocalVariables} \colon \mathsf{LocVarIndex} \rightarrow \mathsf{MemCell},\\
&\quad\ \ \mathsf{OperandStack} \colon \mathsf{list[MemCell]},\\
&\quad\ \ \mathsf{Context} \colon \mathsf{ContextIndex} \,\big]
\end{aligned}
\tag{4.9}
$$

Method is an index into the method component of the CAP file representation pointing to the definition of method that belongs to this frame. It is used to identify the method, associated with this frame, and read more information about the method.

PC is an index into the ByteCodes array of the method that belongs to this frame. The instruction addressed by this index is the instruction that is to be executed next.

Class is an index into the class component of the CAP file representation. The method which is associated with this frame has been invoked as a class member of the class definition addressed by this index. This class is called the current class. This index is needed to identify the class this method belongs to, because the class field in the methodinfo that defines this method points to the class that first defined this method.

LocalVariables is the array of local variables that occur in the method that belongs to this frame. Each local variable is represented by a memory cell. The local variables are indexed by a number starting with 1. The local variable indexed by 0 is used as a reference to *this*, the object this method belongs to.

OperandStack The operand stack is defined as a stack of memory cells. It is used to store results of computations that are not associated with a local variable as well as other values that are needed during execution. Most computations take place on the operand stack with the results being pushed onto it.

Context indicates the context this frame is associated with and represents the active context when this frame is current. The context of a method is determined when the method is invoked and is equal to the context of the object this method belongs to. The currently active context is usually read from the context stack. But since this formalization does not contain a context stack I include it into the frame cell. The active context is used to enforce security mechanisms.

When an applet is initialized at least one frame is created for the method *main*. The applet execution terminates when the method *main* finishes and the respective frame is removed. How frames are created and initialized is described in subsection 5.3 as part of the operational semantics.

4.3. Accessing the State

In order to execute instructions and compute the successor state a JCVM must read data from the current state. The information that is needed for the computation can be data from objects as well as data from the current frame. Since the memory model contains coproducts, accessing the state is more complicated than accessing the CAP file.

The simplest part of the state is the static memory because it does not contain coproducts. Since the range of the static field array is not restricted we could access it directly. We are, however, only interested in reading static fields that actually do exist in an applet. So we have to verify that the static field, that we want to access, actually exists. This is done by comparing its index to the size item of the static field component of the CAP file.

The following example represents the static field *index* in the state s that occurred during execution of CAP file p.

– TYPE THEORY

$$\begin{array}{l}\mathsf{IF}\ index < p.\mathsf{StaticFieldComponent.Size}\ \mathsf{THEN}\\ \quad s.\mathsf{StaticMem}(index)\\ \mathsf{ELSE}\ \ldots\end{array}$$

The else part would define a value for the term in case the static fields is not defined for this CAP file.

The contents of an instance field of an object is represented by a similar term. The test in the first line would compare the index to the FieldCount-item in the class info of the respective class after the existence of the class is verified. The second line then would address the object field of the respective object.

Reading data from the ObjectType-item of the object cell is more complex since the actual type of the object has to be considered. Coproducts therefore have to be investigated with the CASES expression as is shown in the following example which represents the class index of the object referenced by *ref* in the state *s*.

– TYPE THEORY

$$\begin{array}{l}\mathsf{CASES}\ s.\mathsf{HeapMem}(\mathit{ref}).\mathsf{ObjectType}\ \mathsf{OF}\ \big\{\\ \mid \mathsf{Class}\,c \mapsto c\\ \mid \mathsf{Array}\,a \mapsto \ldots\big\}\end{array}$$

For each element of the coproduct a result expression has to be defined. The case of an array is left blanc in this example.

The last example shows how the top element of the operand stack of a the current frame is represented for a given state *s*.

– TYPE THEORY

$$\begin{array}{l}\mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \big\{\\ \mid \mathsf{cons}\,f \mapsto\\ \qquad \mathsf{CASES}\ f.\mathsf{head.OperandStack}\ \mathsf{OF}\ \big\{\\ \qquad \mid \mathsf{cons}\,o \mapsto o.\mathsf{head}\\ \qquad \mid \mathsf{nil}\,() \mapsto \ldots\}\\ \mid \mathsf{nil}\,() \mapsto \ldots\big\}\end{array}$$

The special feature in this example is that the elements of the operand stack are inside two coproducts. Therefore two nested CASES expressions are needed to access the top of the operand stack. The first CASES expression unfolds the stack memory to give access to the current frame. The second CASES expression examines the operand stack and tells us if there exists an element on it. If more than one element from a stack are needed, one CASES expression is used for each element. For the case of an empty stack a term must of course be specified too.

A new state or component of a state is constructed by simply defining the term that represents the state.

5. Operational Semantics

With a representation of the CAP file and the machine memory we can now describe the JCVM in the form of an operational semantics. Like other abstract machines the JCVM inductively applies semantics functions for instructions to a machine state. How these semantics functions are constructed is described in section 5.3. The exception handling mechanism is described in section 5.4. In section 5.5 the semantics function of the Java Card Virtual Machine is defined.

In section 5.2 I will describe the type structure of the operational semantics as a coalgebra. The coalgebraic type structure is special in this formalization and is the basis for the operational semantics of the machine instructions, the exception handling mechanism and the JCVM. First, in 5.1 I will describe how the operational semantics is defined in the the JCVM specification and why we need to describe the JCVM in a formal way.

5.1. Formal vs. Informal Specification

The JCVM can be seen as an abstract machine for executing Java Card bytecode that is defined as a recursive function which repeatedly reads the next instruction from the CAP file and executes it. The semantics of a program can be defined as the fixpoint of execution of the semantics for the next instruction:
$[\![program]\!](state) = fixpoint([\![next\_instruction(state, program)]\!](state, program))$
where the semantics functions for an instruction $[\![instruction]\!]$ has the type $state \times program \rightarrow state$. $state$ is thereby an expression describing a state of the JCVM. The program that is executed is represented by the program-expression $program$.

The JCVM specification [3], which is meant as a guide for implementations of JCVMs, omits formal definitions. The semantics of JCVM instructions are described in a more loose way. For instance, the creation of a new object of a certain class type is described by the JCVM specification as: “Memory for a new instance of that class is allocated from the heap, and the instance variables of the new object are initialized to their default values.” The process of memory allocation is not further described. The layout of class instance objects is roughly described at a different place in the JCVM specification without giving a precise definition.

Furthermore the JCVM specification makes a lot of features optional. Thus, the implementer of a JCVM has a lot of freedom to decide how special aspects of the virtual machine are treated and how to adopt an implementation for its actual purpose. So the goal of the JCVM specification is to keep JCVM implementations compliant to the original specification and thus compatible, so that a CAP file can be safely executed on any virtual machine implementation regardless of the compiler it was created with.

While such informal specifications might be sufficient for guiding an implementation of a virtual machine they may be incomplete and contain ambiguities and inconsistencies. Such inconsistencies are hard to realize and deal with. The main task of virtual machine formalizations is to convert the informal virtual machine description into a formal specification.

Such a formal description can be useful in several aspects: On one hand it can serve as a reference specification for implementation since it defines the operational semantics of a virtual machine much clearer and more accurate than an informal specification. On the other hand it can be used to formally investigate bytecode and the virtual machine itself. For the latter it is useful to utilize a theorem prover in order to use machine support like is done for this formalization.

The rest of this chapter will describe how the informal specification of the JCVM is converted into a formal definition of semantics functions for machine instructions.

5.2. Coalgebraic Type Structure

Before I define the operational semantics of the JCVM, I want to discuss the types of the semantics functions. Other formalizations define a machine state *state* that includes runtime data as well as non runtime data concerning the computation result. Thus the type of the virtual machine just goes from a state to a state: $state \to state$.

In this formalization I choose a different approach to define the JCVM type. The machine state, defined in chapter 4.2, only contains runtime data such as the heap memory and the stack memory. Information about the result of a computation is represented as a coproduct containing an element for each possible outcome. This coproduct is then evaluated separately after every single instruction. The advantage of this approach is that the machine state can be represented by the memory model only.

Metadata about the result which is not stored in the machine state, but represented in the behavior of the machine, is formalized by the *Result*-type

– TYPE THEORY

$$\begin{aligned}
&\mathsf{Result[State, MemCell]} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
&\qquad \{\, \mathsf{Norm} : \mathsf{State} \\
&\qquad \mid \mathsf{Halt_Norm} : \mathsf{State} \\
&\qquad \mid \mathsf{Exception} : [\,\mathsf{state} : \mathsf{State}, \mathsf{excp} : \mathsf{MemCell}\,] \\
&\qquad \mid \mathsf{Halt_Exception} : [\,\mathsf{state} : \mathsf{State}, \mathsf{excp} : \mathsf{MemCell}\,] \\
&\qquad \mid \mathsf{Undef} : \mathsf{State} \,\}
\end{aligned} \tag{5.1}$$

where the components have the following meaning:

Norm indicates that the instruction has been executed successfully. The successor state is returned and the machine is ready to compute the next instruction.

Halt_Norm is returned when the computed instruction caused the machine to terminate normally. This usually happens by returning from the main method. The final state is returned. Computed results may be stored in the object structure on the heap memory. Theoretically it would be possible to restart the machine based on the previous results and continue computation this way, even though the JCVM specification does not define such behavior.

Exception signals that an exception was thrown while attempting to execute the next instruction. The state that caused the exception to be thrown and a pointer to the exception object on the heap is returned.

Halt_Exception is the result in case of an uncaught exception. The heap of the returned state consists of the object structure created during execution. Additionally a reference to the exception instance on the heap is returned which could be used to further evaluate the result. The JCVM specification, however does not define any continuing behavior.

Undef indicates that the semantics of an instruction for a CAP file and the returned state is undefined. The semantics is undefined if the instruction cannot be executed because the CAP file is not well formed. Java Card compilers usually perform sufficient syntax and bytecode checks to ensure that the CAP file is well formed.

Other formalization of virtual machines include an exception flag that indicates a thrown exception as well as halt flags into the machine state. In this formalization such information is entirely represented in the Result-type. Additionally the result distinguishes between machine terminations caused by the program end, an exception or an execution error. So the result can be fully investigated for possible further action.

While the JCVM specification supports a wide range of errors, it only states that the machine halts without any further evaluation of the error that occurred. Since the error does not influence the operational semantics, I decided not to include an element for errors in the Result-type. Instead errors are covered by the undefined result.

Now we can define the type of the JCVM in *JCVMType*.

– TYPE THEORY

$$\begin{aligned}\mathsf{JCVMType} : \mathsf{TYPE} &\stackrel{\text{def}}{=} \\ &\mathsf{State} \times \mathsf{CAPFile} \to \mathsf{Result}[\mathsf{State}, \mathsf{MemCell}]\end{aligned} \tag{5.2}$$

It goes from a state and a CAP file to a result over a state. Since the CAP file is not modified during execution it can be regarded as a constant. Thus the type could be written as $State \to CAPfile \to State + State + State \times MemCell + State \times MemCell + State$ which is in the shape of a coalgebra. See [14] for definitions of the term coalgebra and its use.

The type of JCVM instructions additionally includes the instruction arguments and is described by the following type.

– TYPE THEORY

$$\begin{aligned}\mathsf{InstructionType} : \mathsf{TYPE} &\stackrel{\text{def}}{=} \\ &\mathsf{Argument} \times \ldots \to \mathsf{JCVMType}\end{aligned} \tag{5.3}$$

It goes from the appropriate number of arguments to the JCVM type.

As far as I know, this is the first time that the virtual machine is formalized as a coalgebra. In the PVS formalization the Result-type is simply represented by a data type. Thus no additional tools are necessary to formalize the coalgebraic type in the PVS language. Please note that only the type of the semantics functions are represented as coalgebra.

5.3. Semantics of Instructions

The semantics functions of instructions describe how the machine state is updated during the execution of instructions. By using the type theory we can describe the semantics in a functional style. As described in 5.2 the semantics functions return a term of type Result. An update function for a state s with result Norm is stated as a labeled product update:

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{Norm}\ s\ \mathsf{WITH}\ [\\
\quad (s.\mathsf{HeapMem}) := \mathsf{newHeapMem},\\
\quad (s.\mathsf{HeapTop}) := \mathsf{newHeapTop},\\
\quad (s.\mathsf{StackMem}) := \mathsf{newStackMem},\\
\quad (s.\mathsf{StaticMem}) := \mathsf{newStaticMem},\\
]
\end{array}
$$

newHeapMem,*newStackMem*,*newHeapTop* and *newStaticMem* describe the updated heap memory, stack memory, heap top and static memory respectively.

Before the result of the instruction can be computed, necessary information from the CAP file and the machine state must be read and verified. The CAP file is accessed by the CAP file access functions and memory cell operations only. The state can be accessed directly. However, it is necessary to verify the state structure (e.g. the existence of a top frame on the stack memory or the existence of a local variable in a frame).

Reading and verifying data from the CAP file and machine state is nested in CASES and IF-THEN-ELSE expressions. Some specification languages like ML, that is used in [6], may use pattern matching instead. The type theory language we use doesn't allow pattern matching. The general structure of a semantics function is demonstrated by the nop-instruction which only increases the PC of the current frame.

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{nop}\,]\!] : \mathsf{JCVMType}\\
\quad \stackrel{\text{def}}{=}\ \lambda s\colon \mathsf{State}, p\colon \mathsf{CAPFile}.\\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{\\
\qquad \mid \mathsf{nil}\,()\mapsto \mathsf{Undef}\,s\\
\qquad \mid \mathsf{cons}\,f\mapsto \mathsf{Norm}\,s\ \mathsf{WITH}\ \big[\\
\qquad\quad (s.\mathsf{StackMem}) := \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\,(\mathsf{PC}) = f.\mathsf{head}.\mathsf{PC}+1\,], f.\mathsf{tail})\,\big]\\
\qquad \}
\end{array}
$$

The CASES expression defines a result for each possible stack memory layout. If the stack memory is empty, then the function result is undefined. If a top frame exists, then the new stack memory of the successor state is computed by increasing the program counter of the top frame by one and pushing the new top frame onto the rest stack according to the JCVM specification [3].

The following subsections explain semantics functions that describe the core mechanism of the JCVM like object creation, object access, method invocation, method processing and static field access.

5.3.1. Creating and Accessing Objects

The `new`-instruction which creates an instance of class *class* at the next free heap position is formalized as follows:

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{new}\,]\!] : \mathsf{ClassIndex} \to \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \ \lambda class\colon \mathsf{ClassIndex}, s\colon \mathsf{State}, p\colon \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad \mid \mathsf{cons}\, f \mapsto \\
\qquad\quad \mathsf{CASES}\ \mathsf{getClass}(class, p)\ \mathsf{OF}\ \{ \\
\qquad\quad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\quad \mid \mathsf{up}\, cinfo \mapsto \\
\qquad\qquad \mathsf{CASES}\ cinfo.\mathsf{Typ}\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{Interface} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad \mid \mathsf{Class} \mapsto \\
\qquad\qquad\quad \mathsf{Norm}\, s\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad (\mathsf{HeapMem}(s.\mathsf{HeapTop})) = \big(\mathsf{Type} = \mathsf{Class}\, class, \\
\qquad\qquad\qquad\qquad \mathsf{Field} = \mathsf{initFields}(0, cinfo), \\
\qquad\qquad\qquad\qquad \mathsf{Context} = f.\mathsf{head}.\mathsf{Context}\big), \\
\qquad\qquad\qquad (\mathsf{HeapTop}) = \mathsf{NewReference}(s.\mathsf{HeapTop}), \\
\qquad\qquad\qquad (\mathsf{StackMem}) = \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\quad (\mathsf{OperandStack}) = \mathsf{cons}(s.\mathsf{HeapTop}, f.\mathsf{head}.\mathsf{OperandStack}), \\
\qquad\qquad\qquad\quad (\mathsf{PC}) = f.\mathsf{head}.\mathsf{PC} + 1\,], f.\mathsf{tail}) \\
\qquad\qquad\quad] \\
\qquad\qquad \} \\
\qquad\quad \} \\
\qquad \}
\end{array}
$$

According to the JCVM specification [3] the following preconditions must be verified which is done by using CASES expressions:

- There must be a frame on the frame stack. This test is always required, since at least one frame is necessary to compute something. In case the frame stack is empty, then the behavior is undefined.
- The *getClass* function which is defined in the memory cell operations verifies if the class *class* is defined in the given CAP file and returns the respective classinfo. If the class does not exist then getClass returns a constant *noClassInfo*.
- The type of the class indexed by *class* must be of type Class.

The recursive function *initFields* creates an instance field image *fields* for the new object. It is defined as follows:

– TYPE THEORY

$$\begin{array}{l} \mathsf{initFields}\colon \mathsf{nat} \times \mathsf{ClassInfo} \to [\mathsf{FieldIndex} \to \mathsf{MemCell}] \\ \quad \stackrel{\text{def}}{=} \ \lambda n\colon \mathsf{nat}, cinfo\colon \mathsf{ClassInfo}. \\ \qquad \mathsf{IF}\ n < cinfo.\mathsf{FieldCount}\ \mathsf{THEN}\ \mathsf{initFields}(n+1, cinfo)\ \mathsf{WITH} \\ \qquad\qquad \big[\,(n) = \mathsf{initMemCell}(cinfo.\mathsf{FieldToken}(n))\,\big] \\ \qquad \mathsf{ELSE}\ \lambda n\colon \mathsf{FieldIndex}.\ \mathsf{Null} \end{array}$$

Every instance field in the new object is set to the default value of the type of the instance field by the function *initMemCell* defined in the memory cell operations.

If all preconditions are verified, the successor state is computed by writing the new object of type *class* at the top of the heap. The new object is owned by the context of the current frame. The new heap top is computed with the memory cell operation NewReference. The PC of the current frame is increased by one and the reference to the new object is pushed onto the operand stack of the current frame. The `newarray`-instruction which creates new objects of type array is defined in a similar way.

The example of the `putfields_s_this`-instruction in figure 5.1 shall demonstrate how instance fields are accessed. It sets the instance field n of the object, the current method belongs to, to the top of the operand stack of the current frame. The written instance field must be of type Short and its index is specified as instruction argument n.

The `putfield_s_this`-instruction of the JCVM specification [3] verifies if a current frame exists, if the current method is of type Virtual, if the reference in local variable 0 which contains a pointer to this is of type reference, if the operation is secure, if the current method is allowed to access the this-object and if there is an element on the operand stack of the current method. The functions *accessible* and *secure* are defined as part of the memory cell operations. The function getMethodType is defined as CAP file access function.

The successor state is computed by setting the instance fields specified by this and f to the value of the operand stack top of the current frame. The top of the operand stack is popped from the operand stack of the current frame and the PC of the current method is increased by one.

The JCVM specification [3] defines the instructions `putfield_t`, `putfield_t_this`, `getfields_t` and `getfields_t_this` for accessing Instance fields of class instances where t defines the type of the field to be fetched. All these functions are similar to the `putfield_t_this`-instruction. The fields of array objects and static class fields are accessed by different functions.

5.3.2. Invoking and Processing Methods

Methods can be invoked with `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual`-instructions. The main tasks of method invocation is the creation and

– TYPE THEORY

$$
\begin{array}{l}
[\![\mathsf{putfield\_s\_this}]\!] \colon \mathsf{FieldIndex} \to \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \;\; \lambda n \colon \mathsf{FieldIndex}, s \colon \mathsf{State}, p \colon \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad\; \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\; \mid \mathsf{cons}\, f \mapsto \\
\qquad\qquad \mathsf{CASES}\ \mathsf{getMethodType}(f.\mathsf{head}.\mathsf{Method}, p)\ \mathsf{OF}\ \{ \\
\qquad\qquad\; \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\; \mid \mathsf{Clinit} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\; \mid \mathsf{Virtual} \mapsto \\
\qquad\qquad\qquad \mathsf{CASES}\ \mathsf{getReference}(f.\mathsf{head}.\mathsf{LocalVariables}(0))\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\; \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\; \mid \mathsf{Null} \mapsto \mathsf{throw}(NullPointerException, s, p) \\
\qquad\qquad\qquad\; \mid \mathsf{up}\, ref \mapsto \\
\qquad\qquad\qquad\qquad \mathsf{IF}\ \neg\mathsf{secure}(f.\mathsf{head}.\mathsf{Context}, s.\mathsf{HeapMem}(ref).\mathsf{Context}) \\
\qquad\qquad\qquad\qquad\quad \mathsf{THEN}\ \mathsf{throw}(SecutityException, s, p) \\
\qquad\qquad\qquad\qquad \mathsf{ELSE\ IF}\ \neg\mathsf{accesseble}(n, s.\mathsf{HeapMem}(ref), f.\mathsf{head}.\mathsf{Method}, p) \\
\qquad\qquad\qquad\qquad\quad \mathsf{THEN}\ \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad \mathsf{ELSE\ CASES}\ f.\mathsf{head}.\mathsf{OperandStack}\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}\, o \mapsto \\
\qquad\qquad\qquad\qquad\qquad \mathsf{IF}\ \neg\mathsf{isShort}(o.\mathsf{head}) \vee \neg\mathsf{isShort}(s.\mathsf{HeapMem}(ref).\mathsf{Field}(n)) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{THEN}\ \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad\qquad \mathsf{ELSE}\ \ \mathsf{Norm}\, s\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{HeapMem}(ref).\mathsf{Field}(n) = o.\mathsf{head} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{StackMem}) = \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{OperandStack}) = o.\mathsf{tail}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{PC}) = f.\mathsf{head}.\mathsf{PC} + 1\,], f.\mathsf{head}) \\
\qquad\qquad\qquad\qquad\qquad\quad] \\
\qquad\qquad\qquad\qquad \} \\
\qquad\qquad\qquad \} \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

Figure 5.1.: The semantics of the putfield_s_this-instruction

initialization of a frame on the stack memory for the newly invoked method. The layout of the mentioned method invocation instructions is very similar and is demonstrated by the `invokevirtual`-instruction in figure 5.2.

According to the JCVM specification [3] the `invokevirtual`-instruction verifies the following items:

- There has to be a frame for the current method on the stack memory.
- The arguments of the newly invoked method are stored in the local variables array of the frame for the invoked method by the *pushArgs* function defined in figure 5.3. If there are enough arguments on the operand stack of the invoking method then pushArgs returns a frame for the invoked method and the frame for the current method where the arguments are popped from the operand stack.
- The local variable 0 of the new frame returned by pushArgs contains the *object reference* to the objects that the invoked method belongs to. If the reference to the object is Null, then a NullPointerException is thrown.
- The object referenced by *object reference* is used to resolve the object whose method shall be invoked. This object is verified to be of type Class and the pointer to the class definition in the CAP file is computed. If the object is of type Array, then the class Object would be used which is not defined in this formalization.
- The method addressed by the class and the argument *cm* is resolved by the function *getClassMethod* which is part of the CAP file access functions from section 3.2.
- The *getMethodType* function from the CAP file access functions is used to compute the type of the respective method. The method must exist and be of type Virtual.
- The accessible function from the memory cell operations is used to verify if the current method is allowed to invoke the method.
- The *getMethodContext* function from the CAP file access functions computes the context of the specified method.
- The *secure* function from the memory cell operations verifies if the method invocation operation is secure. If this is not the case, then a SecutityException is thrown

After all preconditions are verified, the successor state is computed by constructing the new stack memory from the new frame for the invoked method, the updated frame from current method where the PC is increased by one and the rest stack.

The JCVM specification [3] does not specify whether the PC of the invoking method is increased by the method invoking instructions or by the return instructions. I decided

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{invokevirtual}\,]\!]\colon \mathsf{MethodIndex} \times \mathsf{nat} \to \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \; \lambda cm\colon \mathsf{MethodIndex}, n\colon \mathsf{nat}, s\colon \mathsf{State}, p\colon \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\; s.\mathsf{StackMem}\; \mathsf{OF}\; \{ \\
\qquad \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad \mid \mathsf{cons}\, f \mapsto \\
\qquad\quad \mathsf{CASES}\; \mathsf{pushArgs}(newFrame, f.\mathsf{head}, n, p)\; \mathsf{OF}\; \{ \\
\qquad\quad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\quad \mid \mathsf{up}\, Frames \mapsto \\
\qquad\qquad \mathsf{CASES}\; \mathsf{getReference}(Frames.\mathsf{new}.\mathsf{LocalVariables}(0))\; \mathsf{OF}\; \{ \\
\qquad\qquad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad \mid \mathsf{Null} \mapsto \mathsf{throw}(NullPointerException, s, p) \\
\qquad\qquad \mid \mathsf{up}\, ref \mapsto \\
\qquad\qquad\quad \mathsf{CASES}\; s.\mathsf{HeapMem}(ref).\mathsf{Type}\; \mathsf{OF}\; \{ \\
\qquad\qquad\quad \mid \mathsf{Array}\, a \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\quad \mid \mathsf{Class}\, class \mapsto \\
\qquad\qquad\qquad \mathsf{CASES}\; \mathsf{getClassMethod}(cm, class, p)\; \mathsf{OF}\; \{ \\
\qquad\qquad\qquad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad \mid \mathsf{up}\, m \mapsto \\
\qquad\qquad\qquad\quad \mathsf{CASES}\; \mathsf{getMethodType}(m, p)\; \mathsf{OF}\; \{ \\
\qquad\qquad\qquad\quad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\quad \mid \mathsf{Clinit} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\quad \mid \mathsf{Virtual}\, token \mapsto \mathsf{IF}\; \neg\mathsf{accessable}(mc, m, token, class, p) \\
\qquad\qquad\qquad\qquad \mathsf{THEN}\; \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad \mathsf{ELSE}\; \mathsf{CASE}\; \mathsf{getMethodContext}(m, p)\; \mathsf{OF}\; \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad \mid \mathsf{up}\, context \mapsto \\
\qquad\qquad\qquad\qquad\quad \mathsf{IF}\; \neg\mathsf{secure}(f.\mathsf{head}.\mathsf{Context}, context)) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{THEN}\; \mathsf{throw}(SecutityException, s, p) \\
\qquad\qquad\qquad\qquad\quad \mathsf{ELSE}\; \mathsf{Norm}\, s\; \mathsf{WITH}\; [\\
\qquad\qquad\qquad\qquad\qquad (\mathsf{StackMem}) = \mathsf{cons}(Frames.\mathsf{new}\; \mathsf{WITH}\; [\\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{Method} = m, \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{PC}) = 0, \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{OperandStack}) = \mathsf{nil}\,(), \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{Context}) = context\,] \\
\qquad\qquad\qquad\qquad\qquad , \mathsf{cons}(Frames.\mathsf{crnt}\; \mathsf{WITH}\; [\\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{PC}) = Frames.\mathsf{crnt}.\mathsf{PC} + 1\,], f.\mathsf{tail})) \\
\qquad\qquad\qquad\qquad\quad] \\
\qquad \}\,\}\,\}\,\}\,\}\,\}\,\}
\end{array}
$$

Figure 5.2.: The semantics of the invokevirtuel-instruction

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{pushArgs}\colon \mathsf{FrameCell} \times \mathsf{FrameCell} \times \mathsf{nat} \times \mathsf{CAPFile} \\
\qquad\qquad\qquad \to \mathsf{Lift}\big[[\,\mathsf{new}\colon \mathsf{FrameCell}, \mathsf{crnt}\colon \mathsf{FrameCell}\,]\big] \\
\quad \stackrel{\mathrm{def}}{=} \; \lambda new, crnt\colon \mathsf{FrameCell}, n\colon \mathsf{nat}, p\colon \mathsf{CAPFile}. \\
\qquad \mathsf{IF}\ n = 0\ \mathsf{THEN}\ \mathsf{up}\,(new, crnt) \\
\qquad \mathsf{ELSE\ CASES}\ crnt.\mathsf{OperandStack}\ \mathsf{OF}\ \big\{ \\
\qquad\quad \mid \mathsf{nil}\,() \mapsto \mathsf{bot} \\
\qquad\quad \mid \mathsf{cons}\, o \mapsto \ \mathsf{pushArgs}(new\ \mathsf{WITH}\ [\,(\mathsf{LocalVariables}(n-1)) = o.\mathsf{head}\,], \\
\qquad\qquad\qquad\qquad\qquad crnt\ \mathsf{WITH}\ [\,(\mathsf{OperandStack}) = o.\mathsf{tail}\,], \\
\qquad\qquad\qquad\qquad\qquad n-1, p) \\
\qquad \big\}
\end{array}
$$

Figure 5.3.: The pushArgs function

to increase the PC by the invoking method to let it consistently point to the instruction which has to be executed next in this method (line of figure 5.2). Thus, the formalizations of the return instructions don't have to increase the PC and become simpler because no check of the caller method is required.

The `invokevirtual`-instruction as defined in the JCVM specification only gets an index into the constant pool as arguments. The constant pool entry at that index contains an item for the number of arguments of the respective method. Since I don't use a constant pool, but need the number of arguments to resolve the invoked method before I can read the number of arguments from the resolved method, I included an argument for the number of arguments in the `invokevirtual`-instruction.

An example of a return instruction is given by the instruction `return` from the JCVM specification [3] which quits the current method without returning any value to the caller method.

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{return}\,]\!]\colon \mathsf{JCVMType} \\
\quad \stackrel{\mathrm{def}}{=} \; \lambda s\colon \mathsf{State}, p\colon \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \big\{ \\
\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\ \mid \mathsf{cons}\, f \mapsto \mathsf{Norm}\, s\ \mathsf{WITH}\ \big[\,(\mathsf{StackMem}) = f.\mathsf{tail}\big] \\
\qquad \big\}
\end{array}
$$

The instruction return just pops the current frame from the stack memory which leaves the frame of the caller method as new current frame. Return instructions that return a value to the calling method push the top of the operand stack of the returning method onto the operand stack of the caller method.

The JCVM specification [3] does not mandate that the return instructions verify the type of the returned value. Therefore such verifications are not included in the formalization of the semantics of these return instructions even though they should be included from a logical point of view.

– TYPE THEORY

$$
\begin{array}{l}
[\![\mathsf{sstore}]\!] \colon \mathsf{LocVarIndex} \to \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \lambda l \colon \mathsf{LocVarIndex}, s \colon \mathsf{State}, p \colon \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\ \mid \mathsf{cons}\, f \mapsto \\
\qquad\qquad \mathsf{CASES}\ f.\mathsf{head}.\mathsf{OperandStack}\ \mathsf{OF}\ \{ \\
\qquad\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\ \mid \mathsf{cons}\, o \mapsto \\
\qquad\qquad\qquad \mathsf{IF}\ \neg\mathsf{isLVIndex}(l, f.\mathsf{head}.\mathsf{Method}, p) \lor \neg\mathsf{isShort}(o.\mathsf{head}) \\
\qquad\qquad\qquad\quad \mathsf{THEN}\ \mathsf{Undef}\, s \\
\qquad\qquad\qquad \mathsf{ELSE}\ \mathsf{Norm}\, s\ \mathsf{WITH}\ \big[\,(\mathsf{StackMem}) = \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad\qquad (\mathsf{LocalVariables}(l)) = o.\mathsf{head}, \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{OperandStack}) = o.\mathsf{tail}, \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{PC}) = f.\mathsf{head}.\mathsf{PC} + 1\,], f.\mathsf{tail}) \\
\qquad\qquad\qquad\qquad\ \big] \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

Figure 5.4.: The semantics of the sstore-instruction

An example of an instruction which processes methods is the `sstore`-instruction given in figure 5.4. The sstore instruction pops the top of the operand stack of the current method and stores the value in the local variable indicated by the argument of the instruction. Before it can do this it has to verify that there is a frame on the stack memory, the argument is a valid index to a local variable in this frame and there is a value on the operand stack of the current frame.

Qian [8] identifies problems with the `jsr` and `ret`-instruction which are used to define subroutines where `jsr` jumps to an instruction inside the current method and `ret` returns to the instruction followed by the respective `jsr`-instruction. The problems are caused by an asymmetry between the jsr that pushes the return address onto the operand stack and `ret` that gets the return address from a local variable. This asymmetry is intended in the JCVM specification [3] and can be used to specify difficult behavior such as nested subroutine calls and recursion. It is possible to return from the current method without returning from some subroutine that is executed, or the virtual machine can skip some subroutines and return to a point several levels below the currently executed subroutine.

According to Qian this causes enormous problems during proofs that involve `jsr` and `ret`. The formalization of the semantics can nevertheless be done without problems. The formalization of the `jsr` instruction is given in figure 5.5 and the `ret` instruction in figure 5.6.

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{jsr}\,]\!] : \mathsf{PCIndex} \to \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \ \lambda newPC : \mathsf{int}, s : \mathsf{State}, p : \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\ \mid \mathsf{cons}\, f \mapsto \\
\qquad\quad \mathsf{CASES}\ \mathsf{getPC}(f.\mathsf{head.PC} + newPC, f.\mathsf{head.Method}, p)\ \mathsf{OF}\ \{ \\
\qquad\quad\ \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\quad\ \mid \mathsf{up}\, pc \mapsto \\
\qquad\qquad \mathsf{Norm}\, s\ \mathsf{WITH}\ \big[\,(\mathsf{StackMem}) = \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\\
\qquad\qquad\quad (\mathsf{OperandStack}) = \mathsf{cons}(\mathsf{toShort}(f.\mathsf{head.PC} + 1), f.\mathsf{head.OperandStack}), \\
\qquad\qquad\quad (\mathsf{PC}) = pc\,], f.\mathsf{tail}) \\
\qquad\qquad \big] \\
\qquad\quad \} \\
\qquad \}
\end{array}
$$

Figure 5.5.: The semantics of the jsr-instruction

Please note that in the JCVM specification the return address value is stored by an extra data-type *returnaddress*. However, since this data-type is only used for these two instructions that are not further used in this formalization and the returnaddress-type doesn't imply special properties, I used the type short instead in this formalization.

5.3.3. Accessing the Static Memory

The static field image is created and initialized by the applet initialization method which is not included in this formalization. There are exactly two kinds of instructions that can

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{ret}\,]\!] \colon \mathsf{LocVarIndex} \to \mathsf{JCVMType} \\
\quad \overset{\mathrm{def}}{=} \;\; \lambda l \colon \mathsf{LocVarIndex}, s \colon \mathsf{State}, p \colon \mathsf{CAPFile}. \\
\qquad\quad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \Big\{ \\
\qquad\quad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\,s \\
\qquad\quad\ \mid \mathsf{cons}\,f \mapsto \mathsf{IF}\ \neg\mathsf{isLVIndex}(l, f.\mathsf{head}.\mathsf{Method}, p)\ \mathsf{THEN}\ \mathsf{Undef}\,s\ \mathsf{ELSE} \\
\qquad\qquad\quad \mathsf{CASES}\ \mathsf{getShort}(f.\mathsf{head}.\mathsf{LocalVariables}(l))\ \mathsf{OF} \\
\qquad\qquad\quad\ \mid \mathsf{bot} \mapsto \mathsf{Undef}\,s \\
\qquad\qquad\quad\ \mid \mathsf{up}\,pc \mapsto \\
\qquad\qquad\qquad \mathsf{Norm}\,s\ \mathsf{WITH}\ \big[\,(\mathsf{StackMem}) := \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [(\mathsf{PC}) = pc], f.\mathsf{tail})\,\big] \\
\qquad\qquad\quad \mathsf{ELSE}\ \mathsf{Undef}\,s \\
\qquad\quad \Big\}
\end{array}
$$

Figure 5.6.: The semantics of the ret-instruction

access the static field image in form of the static memory, the `getstatic_t`-instruction that reads static fields and the `putstatic_t`-instruction that writes static fields. The t indicates the type of the static field to be accessed.

The semantics function of the `putstatic_s`-instruction (figure 5.7) which pushes a short value into a static field shall demonstrate the access of the static memory. The `putstatic_s`-instruction verifies if there exists a frame on the stack memory, if there exists a value on the operand stack of that frame, if the top of the operand stack is of type short and if the index given as argument to this instruction is a valid index into the static memory. After that the top is popped from the operand stack of the current frame and the static field is set to that value. The PC is increased by one. The predicate isStaticIndex is part of the memory cell operations.

All `getstatic` and `putstatic` instructions look very similar. For an unexplained reason the JCVM specification mandates the putstatic instructions to perform security verifications while the getstatic instructions do not. From my knowledge I can not say if this is intended or just an error in the JCVM specification. In the above formalization of the `putstatic_s`-instruction I omitted the security verification.

Static arrays are not represented in the static memory in the JCVM specification [3]. Instead array objects are created by the class initialization methods for each static array in the respective class. These arrays are of course accessed by the respective array access function.

The examples in this section shall be sufficient to describe the general mechanism of

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{putstatic\_s}\,]\!]\colon \mathsf{StaticIndex} \to \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \;\; \lambda t\colon \mathsf{StaticIndex}, s\colon \mathsf{State}, p\colon \mathsf{CAPFile}. \\
\qquad\quad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad\quad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\quad\ \mid \mathsf{cons}\, f \mapsto \\
\qquad\qquad\qquad \mathsf{CASES}\ f.\mathsf{head}.\mathsf{OperandStack}\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\ \mid \mathsf{cons}\, o \mapsto \\
\qquad\qquad\qquad\qquad \mathsf{IF}\ \neg\mathsf{isStaticIndex}(t,p) \vee \neg\mathsf{isShort}(o.\mathsf{head}) \vee \\
\qquad\qquad\qquad\qquad \mathsf{THEN}\ \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad \mathsf{ELSE}\ \ \mathsf{Norm}\, s\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{StaticMem}) = s.\mathsf{StaticMem}\ \mathsf{WITH}\ [\,(t) = o.\mathsf{head}\,], \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{StackMem}) = \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{OperandStack}) = o.\mathsf{tail}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{PC}) = f.\mathsf{head}.\mathsf{PC} + 1\,], f.\mathsf{tail}), \\
\qquad\qquad\qquad\qquad\qquad] \\
\qquad\qquad\qquad \} \\
\qquad\quad \}
\end{array}
$$

Figure 5.7.: The semantics function of the putstatic_s-instruction

the state update and the necessary verifications done by the JCVM instruction. Most instructions are very similar and can be constructed easily and straightforwardly, provided the mechanism is understood and the necessary functions for reading and verifying CAP files and the machine state are available. Please note that the actual verifications are done by functions from the memory cell operations or the CAP file access functions. These functions behave according to the specification of the respective memory cell. Thus, how and if verifications by instructions are made depends on the memory cell formalization and not on the formalization of the operational semantics.

5.4. Semantics of the Exception Handling Mechanism

The JCVM has full support for exceptions. Instead of defining an own exception handling mechanism the JCVM simply refers to the exception handling mechanism defined for the JVM. The program can throw predefined exceptions as well as self defined exceptions. However, the number of predefined exceptions is limited to exceptions that make sense for the JCVM. The predefined exceptions of the JCVM are listed in figure 3.1.

While this formalization includes the exception handling mechanism it does not include the predefined class Throwable which is necessary to derive user defined exceptions. Exception handling in this formalization is mainly needed in the semantics functions of the machine instructions which is described in section 5.3.

5.4.1. Throwing of Exceptions

In the JCVM specification [3] there are two ways of throwing exceptions. First, exceptions can be thrown during the execution of an instruction if necessary requirements for the execution are not being fulfilled. Only the predefined exceptions shown in figure 3.1 can be thrown this way. The function *throw*, which is defined in figure 5.8, is used to throw exceptions by the machine only.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{throw} \colon \mathsf{ClassIndex} \times \mathsf{State} \times \mathsf{CAPFile} \longrightarrow \mathsf{Result[State]}\\
&\quad \stackrel{\text{def}}{=} \lambda e \colon \mathsf{ClassIndex}, s \colon \mathsf{State}, p \colon \mathsf{CAPFile}.\\
&\qquad \mathsf{Exception}\big(\ s.\mathsf{HeapTop},\\
&\qquad\qquad s \ \mathsf{WITH}\ \big[\ (\mathsf{HeapTop}) = \mathsf{NewReference}(s.\mathsf{HeapTop}),\\
&\qquad\qquad\qquad (\mathsf{HeapMem}(s.\mathsf{HeapTop}).\mathsf{Type}) = \mathsf{Class}\, e\,\big], s.\mathsf{HeapTop}\,\big)
\end{aligned}
$$

Figure 5.8.: The function throw

The function throw takes an identifier of a predefined exception e. It creates an object on the heap of the type of the exception identifier and returns the result Exception with the new state and the exception pointer. The exception handler function which is described below is then called by the JCVM that is described in section 5.5.

The second way to throw an exception is the instruction `athrow` which can be used to throw exceptions of any kind in the program bytecode. The semantics of the instruction athrow is defined in figure 5.9.

Unlike the function throw, `athrow` does not create an exception object. Instead the exception object must be created manually. The instruction `athrow` then reads the reference pointing to the exception object from the operand stack top after it verified that an element on the operand stack in the topmost frame exists. The exception object is verified to be a subclass of class Throwable and if the operation is secure. If all requirements are fulfilled, then Exception plus the state and the exception pointer are returned.

– TYPE THEORY

$$
\begin{array}{l}
[\![\,\mathsf{athrow}\,]\!] : \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \ \lambda s\colon \mathsf{State}, p\colon \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\ \mid \mathsf{cons}\, f \mapsto \\
\qquad\qquad \mathsf{CASES}\ f.\mathsf{head.OperandStack}\ \mathsf{OF}\ \{ \\
\qquad\qquad\ \mid \mathsf{nil}\,() \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\ \mid \mathsf{cons}\, o \mapsto \\
\qquad\qquad\qquad \mathsf{CASES}\ \mathsf{getReference}(o.\mathsf{head})\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\ \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\ \mid \mathsf{Null} \mapsto \mathsf{throw}(NullPointerException, s, p) \\
\qquad\qquad\qquad\ \mid \mathsf{up}\, ref \mapsto \\
\qquad\qquad\qquad\qquad \mathsf{CASES}\ s.\mathsf{HeapMem}(ref).\mathsf{Type}\ \mathsf{OF}\{ \\
\qquad\qquad\qquad\qquad\ \mid \mathsf{Array}\, a \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad\ \mid \mathsf{Class}\, class \mapsto \\
\qquad\qquad\qquad\qquad\qquad \mathsf{IF}\ \neg\mathsf{isSubClass}(class, \mathsf{Throwable}, p) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{THEN}\ \mathsf{Undef}\, s \\
\qquad\qquad\qquad\qquad\qquad \mathsf{ELSE\ IF}\ \mathsf{insecure}(f.\mathsf{head.Context}, s.\mathsf{HeapMem}(ref).\mathsf{Context}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{THEN}\ \mathsf{throw}(SecutityException, s, p) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{ELSE}\ \mathsf{Exception}(s, ref) \\
\qquad\qquad\qquad\qquad \} \\
\qquad\qquad\qquad \} \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

Figure 5.9.: The semantics of the athrow-instruction

Note that the instruction `athrow` may throw exceptions itself if the respective preconditions hold, i.e. if the reference pointing to the exception object is a null pointer or if the operation is not certified as secure. These exceptions are then thrown instead of the exception `athrow` was executed for.

An increase of the PC is not required in functions that throw exceptions since the execution of the current method is under no circumstances resumed. The point where the execution is resumed is computed by the functions that determine the code that handles the exception.

5.4.2. Determining the Exception Handler

The exact exception handling mechanism of the JVM or the JCVM has not yet been formalized. This is mostly due to the low importance of the exception handling mechanism for formalizations and verification tasks since it is desired to have a complete representation of the virtual machine instructions first. In [12] the exception mechanism of the Java language is formalized as part of the LOOP project [13].

Instead of defining its own exception handling mechanism, the JCVM specification [3] adopts the exception handling from the JVM specified by the JVM specification [2] without modifications. The exception handling mechanism is started when an exception is thrown either automatically by the JCVM or manually in the Java Card applet with the instruction athrow. An *exception handler* is either a catch or a finally clause. An appropriate exception handler is first searched in the current method. If no appropriate exception handler can be found in the current method, the exception is rethrown in the next method in the method invocation chain and so on until a method with an appropriate exception handler is be found. If no exception handler was be found and the man method is reached, then the JCVM terminates.

The process of searching the method hierarchy is formalized by the function *SearchHandler* shown in figure 5.10 which is called by the JCVM (see section 5.5) when an instruction returned the result Exception.

The function SearchHandler first verifies that the method belonging to the topmost frame on the stack memory exists. The argument that was given to the function SearchHandler must point to the exception object on the heap memory. The class id of the exception object is determined. The exception object can of course not be of an array object. The exception handler that is active at the current instruction for the thrown exception is computed by the function GetExcpHandler which is described below.

If an exception handler could be found in the current method, then the function GetExcpHandler returns the PC at which the catch clause of finally clause starts. The exception reference is pushed onto the operand stack of the current method and the execution continues at the first instruction of the found exception handler.

In case no appropriate exception handler was found in the current method and the top of the method hierarchy is reached the SearchHandler function returns bot and the machine terminates as dictated by the JVM specification [2]. If the method hierarchy top is not reached yet, then the frame of the current method is popped from the stack memory and the exception is rethrown in the next method in the method hierarchy by invoking SearchHandler on the modified machine state. If an appropriate exception handler was found, then SearchHandler returns up containing the state the machine resumes computation with.

The function *GetExcpHandler* (figure 5.11)searches for an exception handler applicable at a certain position in the bytecode of a method. It is formalized recursive over

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{SeachHandler}\colon \mathsf{MemCell} \times \mathsf{State} \times \mathsf{MethodComponent} \to \mathsf{Lift[State]} \\
\quad \stackrel{\text{def}}{=} \ \lambda e\colon \mathsf{MemCell}, s\colon \mathsf{State}, m\colon \mathsf{MethodComponent}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{ \\
\qquad \mid \mathsf{nil}\,() \mapsto \mathsf{bot} \\
\qquad \mid \mathsf{cons}\ f \mapsto \\
\qquad\quad \mathsf{IF}\ f.\mathsf{head.Method} >= m.\mathsf{Size}\ \mathsf{THEN}\ \mathsf{bot} \\
\qquad\quad \mathsf{ELSE}\ \mathsf{CASES}\ s.\mathsf{HeapMem}(e).\mathsf{Type}\ \mathsf{OF}\ \{ \\
\qquad\quad \mid \mathsf{Array}\ a \mapsto \mathsf{bot}, \\
\qquad\quad \mid \mathsf{Class}\ class \mapsto \\
\qquad\qquad \mathsf{CASES}\ \mathsf{GetExcpHandler}(class, p.\mathsf{Methods}(f.\mathsf{head.Method}), f.\mathsf{head.PC}, 0)\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{bot} \mapsto \ \ \mathsf{CASES}\ f.\mathsf{tail}\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{nil}\,() \mapsto \mathsf{bot} \\
\qquad\qquad\qquad\qquad \mid \mathsf{cons}\ f2 \mapsto \\
\qquad\qquad\qquad\qquad\quad \mathsf{SeachHandler}(e, s\ \mathsf{WITH}\ [\,(\mathsf{StackMem}) = f2.\mathsf{tail}\,]\,) \\
\qquad\qquad\qquad\qquad \} \\
\qquad\qquad \mid \mathsf{up}\ h \mapsto \ \ \mathsf{up}\ s\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad (\mathsf{StackMem}) = \mathsf{cons}(f.\mathsf{head}\ \mathsf{WITH}\ [\\
\qquad\qquad\qquad\qquad\quad (\mathsf{PC}) = h, \\
\qquad\qquad\qquad\qquad\quad (\mathsf{OperandStack}) = \mathsf{cons}(e, \mathsf{nil}\,())\,], f.\mathsf{tail})\,] \\
\qquad\qquad \} \\
\qquad\quad \} \\
\qquad \}
\end{array}
$$

Figure 5.10.: The exception handling mechanism

the argument *n*. First the nth exception handler in the given method is computed by the function *GetExceptionInfo* from the CAP file access functions. If no nth exception handler info exists in the method then the function GetExcpHandler returns bot. In case the exception handler info exists it is tested if it is applicable for the current instruction.

The exception handler infos in a method info are sorted in a way, such that the first applicable handler is always the right one. So if the PC of the current method is inside the active range of the exception handler and the exception handler handles exceptions of the given type, then the exception handler start offset into the ByteCodes array is returned. If the exception handler does not handle the given exception, or the PC of the current method is below the active range of the exception handler, then the next exception handler for this method is tested by calling GetExcpHandler again. If the

– TYPE THEORY

$$\begin{aligned}
&\mathsf{GetExcpHandler}\colon \mathsf{ClassIndex} \times \mathsf{MethodInfo} \times \mathsf{PCIndex} \times \mathsf{ExceptionIndex}\\
&\qquad\qquad\qquad \longrightarrow \mathsf{Lift}[\mathsf{ExceptionIndex}]\\
&\stackrel{\text{def}}{=} \lambda e\colon \mathsf{ClassIndex}, m\colon \mathsf{MethodInfo}, pc\colon \mathsf{PCIndex}, n\colon \mathsf{ExceptionIndex}.\\
&\quad \mathsf{CASES\ GetExceptionInfo}(n, m)\ \mathsf{OF}\ \{\\
&\quad \mid \mathsf{bot} \mapsto \mathsf{bot}\\
&\quad \mid \mathsf{up}\, h \mapsto\\
&\qquad\quad \mathsf{IF}\ pc > h.\mathsf{Start} + h.\mathsf{Length}\ \mathsf{THEN\ bot}\\
&\qquad\quad \mathsf{ELSE\ IF}\ pc < h.\mathsf{Start}\\
&\qquad\qquad \mathsf{THEN\ GetExcpHandler}(e, m, pc, n+1)\\
&\qquad\quad \mathsf{ELSE\ IF}\ h.\mathsf{Type} = e\ \mathsf{THEN\ up}\, h.\mathsf{Handler}\\
&\qquad\quad \mathsf{ELSE\ GetExcpHandler}(e, m, pc, n+1)\\
&\quad \}
\end{aligned}$$

Figure 5.11.: The function GetExcpHandler

PC of the current method is above the active range of the exception handler, then no applicable exception handler exists in this method and bot is returned.

5.5. Semantics of the Java Card Virtual Machine

With the semantics of the machine instruction we can now formalize the semantics of the *JCVM* in figure 5.12. Like other abstract machines the JCVM is defined as recursive function that continuously executes the next instruction. The next instruction to be executed by the JCVM is computed by the function GetInstruction defined in 3.2. If the execution of the instruction returns Norm then the JCVM is called on the successor state and the next instruction is executed and so on. The results Undef, Halt_Norm and Halt_Exception terminate the JCVM.

An exception that was thrown during the execution of an instruction must be treated separately. In this case the exception handling function is called and the result is evaluated. The exception handling function SearchHandler may return bot if no appropriate handler could be found for the thrown exception and the machine terminates. If an appropriate exception handler was found, then up is returned and the machine resumes execution with the exception handler.

PVS requires to proof termination for recursive functions. Since the termination of a Java Card program and thus the JCVM cannot be proven, I had to formalize the

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{JCVM} : \mathsf{JCVMType} \\
\quad \stackrel{\text{def}}{=} \ \lambda s : \mathsf{State}, p : \mathsf{CAPFile}. \\
\qquad \mathsf{CASES}\ s.\mathsf{StackMem}\ \mathsf{OF}\ \{\texttt{;; fetch the current frame} \\
\qquad \mid \mathsf{nil}\,() \mapsto \mathsf{Halt\_Norm}\, s \\
\qquad \mid \mathsf{cons}\, f \mapsto \\
\qquad\qquad \mathsf{CASES}\ \mathsf{GetInstruction}(s, p)\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{bot} \mapsto \mathsf{Undef}\, s \\
\qquad\qquad \mid \mathsf{up}\, i \mapsto \\
\qquad\qquad\qquad \mathsf{CASES}\ [\![\, i \,]\!](s, p)\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{Undef}\, r \mapsto \mathsf{Undef}\, s \\
\qquad\qquad\qquad \mid \mathsf{Halt\_Norm}\, r \mapsto \mathsf{Halt\_Norm}\, r \\
\qquad\qquad\qquad \mid \mathsf{Halt\_Exception}\, e \mapsto \mathsf{Halt\_Exception}\, e \\
\qquad\qquad\qquad \mid \mathsf{Norm}\, r \mapsto \mathsf{JCVM}(r, p) \\
\qquad\qquad\qquad \mid \mathsf{Exception}\, e \mapsto \ \ \mathsf{CASES}\ \mathsf{SeachHandler}(e.\mathsf{excp}, e.\mathsf{state}, p)\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{bot} \mapsto \mathsf{Halt\_Exception}(e.\mathsf{state}, e.\mathsf{excp}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{up}\, r \mapsto \mathsf{JCVM}(r, p) \\
\qquad\qquad\qquad\qquad\qquad\qquad \} \\
\qquad\qquad\qquad \} \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

Figure 5.12.: The semantics of the JCVM

JCVM semantics as an inductive function. However, in PVS inductive functions are only allowed to return boolean types. The function *Run* which represents the JCVM in the PVS specification therefore is of type $State \times CAPFile \rightarrow boolean$ and returns *true* if and only if the Java Card program represented by the given CAP file terminates normally. The semantics function of instructions in the PVS specification is computed by the function *Semantics* which takes an instruction as argument and calls the respective semantics function.

While the above function describes the operational semantics of the JCVM in a functional style, sometimes it is desired to describe the semantics of the JCVM by means of state transitions. Thus, we want to know if a program p is a model for the transition of state *s1* to state *s2* in one step: $p \vdash s1 \rightarrow s2$. This relation is described by the predicate *Arrow*.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{Arrow}\colon \mathsf{State} \times \mathsf{State} \times \mathsf{CAPFile} \to \mathsf{boolean}\\
&\stackrel{\text{def}}{=} \lambda s1, s2\colon \mathsf{State}, p\colon \mathsf{CAPFile}.\\
&\qquad (s1.\mathsf{StackMem} = \mathsf{cons}\, f)\\
&\qquad \wedge (\mathsf{GetInstruction}(p.\mathsf{MethodComponent}, f.\mathsf{head.PC}, f.\mathsf{head.Method}) = \mathsf{up}\, i)\\
&\qquad \wedge [\![\, \mathsf{i}\,]\!](s1, p) = \mathsf{Norm}\, s2
\end{aligned}
$$

The structure of the machine state, the CAP file and the coalgebraic result require several tests on the CAP file and the result. The instruction which has to be executed next must exist and the computation result must be Norm. Results different to Norm are not considered by this predicate since they may not represent a state transition or are not interesting for this predicate.

With the function Arrow we now can define the state transition in arbitrary steps: $p \vdash s1 \to^* s2$ as predicate *Arrow_Star* in the usual way.

– TYPE THEORY

$$
\begin{aligned}
&\mathsf{Arrow\_Star}\colon \mathsf{State} \times \mathsf{State} \times \mathsf{CAPFile} \to \mathsf{boolean}\\
&\stackrel{\text{def}}{=} \lambda s1, s2\colon \mathsf{State}, p\colon \mathsf{CAPFile}.\\
&\qquad s1 = s2 \vee \exists s3\colon \mathsf{State}.\, (\mathsf{Arrow}(s1, s3, p) \wedge \mathsf{Arrow\_Star}(s3, s2, p))
\end{aligned}
$$

The predicate Arrow_Star is defined recursively by existence of a state *s3* where $p \vdash s1 \to s3$ and $p \vdash s3 \to^* s2$ must hold. The case where *s1* is equal to *s2* must be treated separately since it is not covered by the existence clause.

6. Examples

In this chapter I will discuss some properties of this formalization by giving some program examples and examples for verifications. Section 6.1 will explain how a virtual machine formalization can be used for verifications in general by showing simple examples of verifications of CAP file properties as well as properties of the operational semantics.

The difference between the typed and untyped memory cell formalizations from section 4.1 will be discussed in section 6.2 by giving a bytecode example and by proving properties of both memory cell versions.

In section 6.3 I will discuss the difference between bytecode and source-code verification with an example Java class. A formalization of the Ackerman function is described in section 6.4 to show the expression capabilities of the formalization.

6.1. Verification Examples

One of the main profits of a formal specification is the possibility to investigate an object with formal techniques. Properties of the specified objects can be proven and errors in the specification can be determined. In a JCVM formalization there are 2 main targets for verifications, the bytecode and the operational semantics. Both will be discussed briefly in the following two subsections.

All examples in this chapter are given as PVS source only. However, it should be possible to understand the examples since the PVS language is very intuitive and similar to the type theory language defined in section 2.3 that was used in the previous chapters. The examples state simple properties using predicate logic.

The purpose of the examples in this chapter is to show some aspects, capabilities and possible use of this formalization and formalizations in general. The examples are not intended to really investigate and verify this formalization or CAP files. So all examples are kept simple and are not meant to state significant properties.

More examples of how formalizations are used can be found in the literature especially in [7], [8], [10] and [17].

6.1.1. Bytecode Properties

In chapter 3.2 the type of the CAP file is defined. An instance of this type can be regarded as the formalization of a particular CAP file or a representation of a Java Card program in bytecode form. Thus, we could build a representation of a real CAP file by converting it into the type theory notation or the language of a theorem prover. Properties of Java Card bytecode can be formalized as well and proven to hold for a particular CAP file. Such properties are formalized as predicates taking an instance of type CAPfile as parameter.

An example of bytecode properties is the predicate *MethodCalls_not_undefined* which states that all method invocations in a CAP file invoke a method that actually exists.

PVS

```
MethodCalls_not_undefined (p : CAPFileType) : bool =
FORALL (m:Index) : ((m >= 0) AND (m < p`MC`size)) IMPLIES
  (FORALL (pc:Index): ((pc >= 0) AND (pc < p`MC`method(m)`maxPC)) IMPLIES
    (isinvokevirtual(instr) IMPLIES
      ( (m1(instr) >= 0) AND (m1(instr) < p`MC`size) )
    ) WHERE instr = p`MC`method(m)`Code(pc)
  )
```

This is done by stating that if an instruction invokes a method, then the method which is invoked by that instruction is inside the defined range of the defined methods in the method component of the CAP file. The instruction is all-quantified over all defined instructions in all defined methods in the CAP file.

The predicate MethodCalls_not_undefined only considers the instruction *invokevirtual* since it is the only method invocation function which has been formalized in this thesis. In a complete formalization all method invocation functions must be considered.

This predicate is simply proved with a case analysis which checks every single instruction in the bytecode. It could of course be done by just looking carefully at the bytecode, thus this predicate doesn't do anything which couldn't be done without a formal specification. The advantage of utilizing a machine for this verification is the speed in which it does the proof. Also nobody wants to check thousands of instructions by hand. In PVS this predicate can be proven with a single *GRIND*-command, a predefined proof strategy. No manual case-analysis is necessary. The verification of actual bytecode examples are discussed in sections 6.3 and 6.4.

The previous example states a structural property of the bytecode only. How the single instructions are executed is not considered. Thus the predicate MethodCalls_not_undefined defines a *static property*. It is, however, possible to use the operational semantics to state and prove properties of bytecode. Such properties are called *dynamic properties*.

The predicate *HeapTop_Keeps_Const* states the dynamic property that the heaptop keeps constant during execution.

PVS

```
HeapTop_Keeps_Const (p : CAPFileType) : bool =
FORALL (m:Index) : ((m >= 0) AND (m < p`MC`size)) IMPLIES
  (FORALL (pc:Index): ((pc >= 0) AND (pc < p`MC`method(m)`maxPC)) IMPLIES
    (isNorm(res) IMPLIES (newState`HeapTop = State`HeapTop
                          ) WHERE newState = NormState(res)
    ) WHERE res=Semantics(p`MC`method(m)`Code(pc))(State, p)
  )
```

It all-quantifies over all JCVM states and all defined instructions in a given CAP file. Then it says that if an instruction executes to Norm then the top of the heap memory after the execution is equal to the top of the heap before the execution. The function `Semantics` is the PVS-function that calls the semantics function of the instruction given as first argument to it.

It is clear that this property is only true if the specified bytecode is not able to generate or delete objects on the heap. This property ,of course, does not hold for all programs since some programs might generate class instances or arrays. It can however be proven to hold for the bytecode examples in section 6.3 and 6.4 with the proof command GRIND. The more general property that the heap memory either stays constant or increases is discussed in the next section as an example of a JCVM property.

By stating static and dynamic bytecode properties that must hold for save execution of a CAP file one could define a bytecode verifier. Proving these properties for a particular CAP file would then prove that it is save to execute the CAP file. However, the JCVM specification [3] does not define a bytecode verifier like the JVM specification [2].

The bytecode verifier can of course also be formally investigated as is done by Pusch [7]. Pusch formalizes the bytecode verifier of the JVM and proves that if a state s and a Java class file c are both well formed and the JVM specification executes s and c to a result state s', then the s' is well formed too.

6.1.2. JCVM Properties

The predicate HeapTop_Keeps_Const in the previous example uses the operational semantics to show a property of a certain CAP file. If we all-quantify the predicate over all CAP files, then we would not be able to prove it since the resulting property would not hold. However, if we changed the predicate to state that the top of the heap increases or keeps constant then we can prove the property for all CAP files. This property is stated as predicate *HeapTop_Increases*.

— PVS

```
HeapTop_Increases (p : CAPFileType) : bool =
FORALL (m:Index) : ((m >= 0) AND (m < p`MC`size)) IMPLIES
  (FORALL (pc:Index): ((pc >= 0) AND (pc < p`MC`method(m)`maxPC)) IMPLIES
    (isNorm(res) IMPLIES (newState`HeapTop >= State`HeapTop
                         ) WHERE newState = NormState(res)
    ) WHERE res=Semantics(p`MC`method(m)`Code(pc))(State, p)
  )
```

This predicate is independent of the CAP file which shall be executed. Thus, it doesn't state a property of a CAP file but of the operational semantics itself. It states that the heaptop cannot decrease and thus no objects on the heap memory can be erased by any program. This is indeed the intended behavior of the JCVM since no garbage collection is specified for the JCVM and no instruction is able to erase objects. This predicate is as well completely proven with the PVS-proof-command GRIND.

By all-quantifying over the CAP file we are able to state and prove properties of the JCVM. So one can prove that the JCVM specification implies certain intended behavior and does not contain errors that it should avoid. Also it is possible to test the behavior of the JCVM under special circumstances.

The predicate *StackMem_must_be_nonempty* states that only JCVM states with a non-empty stack memory can be executed to Norm.

— PVS

```
StackMem_must_be_nonempty : LEMMA
FORALL (s:JCVMState,p:CAPFileType,i:Instruction) :
  (isNorm(res) IMPLIES
    (s`StackMem /= null)
    ) WHERE res = Semantics(i)(s,p)
```

It all-quantifies over the states and the CAP files and says that if the state and the CAP file execute to Norm, then the stack memory of the state was not empty.

Again this is intended behavior of the JCVM and can be proven with the PVS-proof-command GRIND. This examples shall be sufficient to show how we can investigate the JCVM with this formalization. More complex properties can easily be specified those proves might be more difficult. Some more complex properties are discussed in the next section.

6.2. Typed vs. Untyped Memory Cell

In section 4.1 I describe how the abstraction of the memory cell is used to model different levels of defensiveness of the JCVM. The two examples of memory cell implementations that are given in subsections 4.1.2 and 4.1.3 formalize a non-defensive and defensive JCVM respectively. The two implementations have of course different behavior. This issue shall be demonstrated and discussed in the following two subsections.

6.2.1. Behavioral distinction

Consider the following bytecode sequence which is taken from the example program in the appendix A.1.

```
1       sconst(1)
2       putfield_s_this(0)
3       new(0)
4       sinc(0,1)
5       putfield_s_this(0)
```

Instruction (1) pushes the short value 1 onto the operand stack which is in (2) pushed into the field 0 of the this-object. Instruction (3) creates a new class instance of class 0 which results in the address of the new object being pushed onto the operand stack. In (4) the local variable 0 which contains the pointer to the this-object is increased by 1. Instruction (5) pushes the top of the operand stack which contains the address of the newly created object into the field 0 of the this-object. The pointer of the this-object however was increased by 1 and now points to a different address in the heap memory.

The idea of the program is to use the `putfield_s_this`-instruction to write a field of a newly created object. After increasing the this pointer by 1 in (4) the this pointer should point to the newly created object, at least in this formalization. However, it is clear that this strategy is terribly wrong. Neither the this-pointer in local variable 0 should be written by the program, nor should fields of an object different to the this-object be written with the putfield_s_this instruction.

The error is the `sinc`-instruction in (4) which is used to increase the local variable 0 which contains a value of type reference. The `sinc`-instruction can only be used on values of type short. Actually there is no instruction to increase reference values defined by the JCVM specification [3].

The typed memory cell formalization is able to detect this error and return Undef (see section 5.2) which causes the JCVM to terminate. The untyped version, however, does not store type information and is not able to detect this error. Since values of type short and reference are represented in the same way, the untyped formalization indeed increases local variable 0 and writes into the new object in (5).

Thus, this example demonstrates two aspects of the different formalizations. First, they behave indeed differently as we could see. Second, the untyped version can perform as intended even though the bytecode contains conceptional errors. Such errors can of course cause the JCVM to perform in unintended ways too. This is a significant problem, since the JCVM may be implemented non-defensive. Errors contained in the bytecode could not be detected by just running the program and later cause the program to misbehave.

Such errors are generated by the Java compiler which should verify the source code and generate valid bytecode. However, we can not be sure if the compiler always generates valid bytecode. Thus, checking programs with defensive JCVM implementations or bytecode verifiers which could detect such errors is necessary to guarantee that the bytecode loaded onto Java cards is valid.

6.2.2. Formal distinction

In subsection 6.1.2 I described how we can show properties and behavior of the JCVM in formal proofs. Since the different formalizations of memory cells in section 4.1 behave differently, we are able to show the behavior of the different formalizations and their differences in formal proofs too.

The following predicate *Class_Types_Keep_Sane* states that the execution of an instruction can not generate a result Norm with class instances on the heap that do not have a class definition in the CAP file provided there have been no class instances on the heap of undefined classes in the previous state.

PVS

```
Class_Types_Keep_Sane : LEMMA
  FORALL (s:JCVMState,p:CAPFileType,i:Instruction) :
  (isNorm(res) IMPLIES (FORALL (ob:TMC):
         ((isClass(s`HeapMem(ob)`Typ) IMPLIES
          (class(s`HeapMem(ob)`Typ)<p`CC`size))
      IMPLIES
          (isClass(news`HeapMem(ob)`Typ) IMPLIES
          (class(news`HeapMem(ob)`Typ)<p`CC`size)))
    ) WHERE news = NormState(res)
  ) WHERE res = Semantics(i)(s,p)
```

This predicate can be proven for the typed memory cell formalization with the PVS-proof-command GRIND. The predicate is true, since the typed memory cell formalization verifies if newly generated class instance have a definition in the class component of the CAP file. Thus, a defensive JCVM can not generate class instances without a class definition available.

This property does not hold for the untyped memory cell formalization. A non-defensive JCVM does not verify if a class from which a new class instances is generated exists in the class component. A `new`-instruction can have a parameter of a non-existing class and still not cause the JCVM to return a result which is not Norm.

The predicate *Class_ Types_ Get_ Unsane* is the negated predicate Class_Types_Keep_Sane and does hold for the untyped memory cell formalization.

PVS

```
Class_Types_Get_Unsane : LEMMA
  NOT (FORALL (s:JCVMState,p:CAPFileType,i:Instruction) :
  (isNorm(res) IMPLIES (FORALL (hp:UTMC):
         ((isClass(s`HeapMem(ob)`Typ) IMPLIES
          (class(s`HeapMem(ob)`Typ)<p`CC`size))
      IMPLIES
          (isClass(news`HeapMem(ob)`Typ) IMPLIES
          (class(news`HeapMem(ob)`Typ)<p`CC`size)))
    ) WHERE news = NormState(res)
  ) WHERE res = Semantics(i)(s,p))
```

This predicate states that the JCVM can generate a state which contains class instances without definition in the CAP file from a state that does not.

The proof of this predicate is more difficult than the proof of predicate Class_Types_Keep_Sane and can not be done with the PVS-proof-command GRIND only. The variables s, p and i have to be instantiated manually since the prover can not guess these correctly. The state s must be set to a state where all elements on the heap are defined and class instances are defined in the given CAP file p. The CAP file p can be chosen from the examples in the appendix. The instruction i has to be a `new`-instruction with a parameter, that is greater than the size item in the class component of CAP file p. The untyped memory cell hp which represents pointers into the heap memory is instantiated to the top of the heap memory in the given state s. With this instantiation the function Semantics would generate a class instance on the top of the heap without a definition in the CAP file p and the predicate is proven to be true.

The formal investigation proofs that the different memory cell formalizations have indeed complementary properties and behavior. These properties can be identified and proven for the single formalizations. Thus, we are able to choose the right implementation of a JCVM for a particular purpose and justify our choice formally. The purpose of the JCVM implementation is formalized as its properties in the shape of predicates. The previous example could be the part of the formalization of the property that the JCVM does manage the heap memory correctly. The right choice would be a defensive JCVM implementation as specified by the typed memory cell abstraction from subsection 4.1.3.

6.3. Source-code vs. Bytecode Verification

The following example is intended to give an idea of how source code and bytecode verification relate. For that I will compare the verification of a Java class with the verification of its bytecode representation. Consider the Java class Logic which was originally proposed by Bergstra [18] to demonstrate side-effects of boolean logic in the Java language.

— JAVA

```
class Logic {
  boolean b, r1, r2;      // r1, r2 will store the result
  boolean f() { b = !b; return b; }
  void m() { b = true;
             r1 = f() || !f();
             r2 = !f() && f(); }
}
```

The example was originally used to show that side-effects can disrupt the expected behavior of logic disjunction and conjunction. Usually we would expect ($\varphi \vee \neg\varphi = \mathsf{true}$) and ($\neg\varphi \wedge \varphi = \mathsf{false}$). However, after running method `m()`, the instance field `r1` will be false and instance field `r2` will be true.

The class Logic was verified by Jacobs [10] using the LOOP tool [13]. After compiling it with the LOOP compiler into a PVS representation the class was verified. On a Pentium II/500MHz/265MB RAM the proof lasts about 70 seconds.

I wrote a bytecode representation which resembles the functionality of the class Logic in appendix A.2 as `SideEffects`. The bytecode representation was written by hand since there is no compiler available which converts a CAP file into its PVS representation. Neither was it possible to convert a CAP file by hand, since the JCVM formalization lacks some necessary parts like class initialization to formalize a real CAP file.

In the bytecode formalization the method `m()` is formalized by the method indexed by 0. The method `f()` which negates the object field `b` and returns the result is formalized by the method indexed by 1. The fields `b`, `r1` and `r2` are the object fields indexed by 0, 1 and 2 respectively. The boolean values true and false are represented by the values 1 and 0 of type short. The Java Card language does not contain the type boolean. A static field component does not have to be specified since the class does not deal with static fields. The class Logic itself is formalized in the class component by class indexed by 0.

The JCVM formalization in PVS can now be used to execute the method `m()` of the class Logic by proofing the lemma `Run (InitState (NullState), SideEffects)` with the strategy `(REPEAT (THEN (EXPAND "Run") (GRIND)))`. The function `InitState` is used to prepare a state for execution by creating a class instance of the class indexed by 0 and a method frame for the method indexed by 0. `NullState` is a constant minimal state. The function `Run` is the PVS formalization of the function JCVM which returns true if and only if the execution of the given CAP file terminates with Halt_Norm. Thus, the lemma proves proper termination of a method in the CAP file. The proof would of course not terminate if the CAP file would not terminate.

After the proof is completed the result of the computation can be seen in the heap memory of the final state. As we would expect, the field 0 which represents field `b` is 1, field 1 which represents `r1` is 0 and field 2 which represents field `r2` in the class Logic. On a machine with comparable power to the machine that was used for the source code verification the proof lasts about 400 seconds.

It is obvious that the verification of the bytecode takes significantly more time than the verification of the source code. There are two reasons for this difference. First, the bytecode contains much more instructions that have to be executed than the source code contains commands. Thus, the bytecode is more complex than the corresponding source code. While Java commands and bytecode instructions can not really be compared the difference is large enough to see that the verification of the source code takes less time. The second reason is the specification of the operational semantics. While the bytecode has to be executed instruction by instruction, the semantics of the Java language allows the LOOP project to implement fancy mechanisms that rewrite the source-code-string to the result instead of stepwise executing commands.

The operational semantics of bytecode instructions presented in chapter 5 show that the semantics of the bytecode instructions is very complex. The semantics functions additionally use helper functions from the memory cell abstraction defined in section 4.1.1. Therefore the bytecode verification is very complex and time consuming. Even with a complete representation of the JCVM it will be difficult and expensive to verify interesting properties of real-life applications.

6.4. Ackerman Function

The last example shall demonstrate the expressive power of the JCVM specification and this formalization. The Ackerman function in figure 6.1 is used as standard example for the over-exponential function class and non-primitive recursion.

While the Ackerman function has no real-life relevance it is often used to show capabilities of compilers and computer languages. Figure 6.2 shows a Java class containing a method that implements the Ackerman function. The function main then computes

$$ack\,0\,n = n+1 \tag{6.1}$$
$$ack\,(m+1)\,0 = ack\,m\,1 \tag{6.2}$$
$$ack\,(m+1)\,(n+1) = ack\,m\,(ack\,(m+1)\,n) \tag{6.3}$$

Figure 6.1.: The Ackerman function

— JAVA

```java
class Ackerman {
 public static void main(String[] args) {
  int a;
  a=ack(2,2);
 }
 static int ack(int x,int y) {
  if (x==0) return y+1;
  if (y==0) return ack(x-1,1);
  return ack(x-1,ack(x,y-1));
 }
}
```

Figure 6.2.: The Java class Ackerman

ack 2 2 which equals 7.

In the appendix A.3 a bytecode formalization of the class in figure 6.2 is given. The bytecode was written by hand and resembles the functionality of the corresponding Java class instead of being generated from the Java class by an actual compiler. In the method component of the bytecode representation the method indexed by 0 represents the main method and method indexed by 1 represents the method `ack()`. The class `Ackerman` is indexed by 0 in the class component.

This bytecode formalization can be used to compute the Ackerman function with the JCVM formalization. The result of the computation with the JCVM formalization can be found in the local variable 0 of the only object on the heap memory. Thus, we are able to formalize and verify over-exponential and non-primitive recursive functions.

However, the computation of *ack* 2 2 with the JCVM formalization takes about one day on my machine which also shows the complexity limitations of the formalization. The main reason, why the computation takes much longer than the other bytecode examples is the size of the stack memory. Since the Ackerman function is defined recursive there are up to 9 frames on the frame stack during the computation of *ack* 2 2. The stack memory is represented in PVS by a datatype. The handling of large data-types seems to generally slow down proofs with PVS.

The computation of *ack* 3 3 would be too complex for the formalization and is not possible at least on my machine. The Java class is capable of computing *ack* 3 3 and returns 61. *ack* 4 4 is too hard for both, the Java program as well as the bytecode formalization.

The predicates MethodCalls_not_undefined and HeapTop_Keeps_Const can be proven for the bytecode formalization of the class Ackerman in appendix A.3 with the PVS-proof-command GRIND. The predicate MethodCalls_not_undefined takes 15 seconds, the predicate HeapTop_Keeps_Const takes 700 seconds. Again the proof HeapTop_Keeps_Const takes significantly more time since the formalization of the operational semantics is involved.

7. Concluding remarks

The importance of the verification of software products to ensure proper functioning, especially in the area of software security, is undisputed. The complexity of such tasks, however, makes the actual proof of respective properties a difficult and expensive effort. The goal of this work was to develop an approach to formalize and formally investigate the Java Card virtual machine. In conclusion a resume of this work and an outline of future work shall be given.

7.1. Resume

In this thesis I presented an approach to formalize the Java Card virtual machine using higher order logic. It is shown, that such a formalization is possible and examples of basic concepts contained in the JCVM semantics are given. The state based approach includes a representation of the machine memory and the CAP binary file that is processed by the virtual machine.

As far as I know, the coalgebraic representation of the type structure of the operational semantics and the abstraction of the memory cell have not previously been presented like in this thesis. The coalgebraic type structure is shown to be suitable for describing the operational semantics and allows handy use of the operational semantics when specifying and verifying JCVM properties. The memory cell abstraction is proven to be a powerful way to specify different JCVM behavior while reusing other parts of the formalization without modification. It makes it a flexible JCVM representation that can be modified easily and adapted to specific environments.

The higher order logic of the PVS tool that was used in the practical part of the work has proven to be suitable for formal specifications of this kind. Also, the PVS proofer was able to cope with the formalization and the presented example verifications.

The proof examples in chapter 6 show that the formalization can be used for several problems. For particular representations of CAP files we can verify the well formedness of the bytecode. For the operational semantics of the JCVM we can state and prove proper behavior. An extensive investigation of the verification capabilities, as well as a complete specification of the variety of verification problems this formalization can be used for, remains outside the scope of this work.

7.2. Future Work

In its current state the formalization is hardly usable for real-life verifications. Several crucial parts of the Java Card platform still need to be formalized or completed including:

- Standard Java Card API classes are necessary to execute Java Card bytecode. Such classes are for instance the classes `Object` and `Throwable` that are of course needed for verifications too.
- The Java Card Runtime Environment is needed to perform CAP file linking, applet initialization and communication with the smart card reader. Without these parts of the JCRE CAP files can neither be executed not verified. Especially the security mechanisms of the JCRE that are only sketched in this work could be an important part and target for verifications.
- Since real-life Java Card packages can hardly be translated by hand into a PVS representation a tool that does this task is needed. Further the current representation of the CAP file which differs in some aspects from the CAP file specification should be verified to be a valid representation of the CAP file.
- Unlike the verification examples presented in this work, properties that have relevance in real life can usually not be carried out using predefined proof strategies. Thus, ways for stating more complex properties about CAP files and the operational semantics and strategies that can proof these properties have to be developed.
- Proofs with the current formalization are very slow. It might be possible to increase the performance by developing clever proof strategies.

A. The PVS Theories

A.1. Types vis. Untyped Memory Cell

PVS

```
Methods : MethodComponent = (#
  Size     := 1,
  Methods  := LAMBDA (m:MethodIndex | m<1) : (#
    nArgs      := 1,
    Args       := LAMBDA (x:nat) : Reference,
    Class      := 0,
    maxPC      := 6,
    maxLV      := 0,
    Typ        := Virtual(none),
    ExceptionCount := 0,
    Exceptions:= LAMBDA (x:ExcpIndex) : noExcpInfo,
    Context    := 1,
    ByteCodes := LAMBDA (pc:PCOffset | pc<6): COND
      pc=0 -> sconst(1),
      pc=1 -> putfield_s_this(0),
      pc=2 -> new(0),
      pc=3 -> sinc(0,1),
      pc=4 -> putfield_s_this(0),
      pc=5 -> return,
      ELSE-> noInstr  ENDCOND
     #)
  #)

Classes : ClassComponent = (#
  Size    := 1,
  Classes := LAMBDA (c:ClassIndex | c<1) : (#
    Typ         := Class,
```

```
    Context     := 1,
    MethodCount := 2,
    MethodTable := LAMBDA (n:MethodIndex|n<2):1,
    SuperClass  := noSuperClass,
    FieldCount  := 3,
    FieldTable  := LAMBDA (n:FieldIndex | n<3) :
                   (# Token:=none, Typ:=Short#)
    #)
  #)

TypedVsUntypes : CAPFile = (#
   MC := Methods,
   CC := Classes,
   SC := noStaticComponent
  #)
```

A.2. Side-effects of Boolean Logic

— PVS

```
Methods : MethodComponent = (#
  Size := 2,
  Methods := LAMBDA (m:MethodIndex | m<2) : COND
    m = 0 -> (#
             nArgs   := 1,
             Args    := LAMBDA (x:nat) : Reference,
             Class   := 0,
             maxPC   := 17,
             maxLV   := 0,
             Typ     := Virtual(none),
             ExceptionCount := 0,
             Exceptions    := LAMBDA (x:ExcpIndex) : noExcpInfo,
             Context := 1,
             ByteCodes    := LAMBDA (pc:PCOffset | pc<17): COND
                        % b = true;
                        pc = 0  -> sconst              (1),
                        pc = 1  -> putfield_s_this     (0),
                        % r1 = f() || !f();
```

```
                     pc = 2  -> aload                (0),
                     pc = 3  -> invokevirtual        (1,1),
                     pc = 4  -> aload                (0),
                     pc = 5  -> invokevirtual        (1,1),
                     pc = 6  -> sneg                 ,
                     pc = 7  -> sor                  ,
                     pc = 8  -> putfield_s_this      (1),
                     % r2 = !f() && f();
                     pc = 9  -> aload                (0),
                     pc = 10 -> invokevirtual        (1,1),
                     pc = 11 -> sneg                 ,
                     pc = 12 -> aload                (0),
                     pc = 13 -> invokevirtual        (1,1),
                     pc = 14 -> sand                 ,
                     pc = 15 -> putfield_s_this      (2),
                     pc = 16 -> return               ,
          ELSE-> noInstr  ENDCOND
          #),
    m = 1 -> (#
          nArgs      := 1,
          Args       := LAMBDA (x:nat) : Reference,
          Class      := 0,
          maxPC      := 5,
          maxLV      := 0,
          Typ        := Virtual(none),
          ExceptionCount := 0,
          Exceptions:= LAMBDA (x:ExcpIndex) : noExcpInfo,
          Context    := 1,
          ByteCodes := LAMBDA (pc:PCOffset | pc<5): COND
                     % boolean f() { b =!b; return b; }
                     pc = 0  ->  getfield_s_this     (0),
                     pc = 1  ->  sneg                ,
                     pc = 2  ->  putfield_s_this     (0),
                     pc = 3  ->  getfield_s_this     (0),
                     pc = 4  ->  sreturn             ,
                     ELSE-> noInstr   ENDCOND
          #),
    ELSE -> noMethod
    ENDCOND
    #)

Classes : ClassComponent = (#
```

```
  Size := 1,
  Classes        := LAMBDA (c:ClassIndex | c<1) : (#
    Typ          := Class,
    Context      := 1,
    MethodCount  := 2,
    MethodTable  := LAMBDA (n:MethodIndex|n<2):1,
    SuperClass   := noSuperClass,
    FieldCount   := 3,
    FieldTable   := LAMBDA (n:FieldIndex | n<3) :
                      (# Token:=none, Typ:=Short#)
    #)
  #)

SideEffects : CAPFile = (#
   MC := Methods,
   CC := Classes,
   SC := noStaticComponent
   #)
```

A.3. Ackerman Function

PVS

```
Methods : MethodComponent = (#
  Size := 2,
  Methods := LAMBDA (m:MethodIndex | m<2) : COND
    m=0 ->  (#
            nArgs     := 1,
            Args      := LAMBDA (x:nat) : Reference,
            Class     := 0,
            maxLV     := 1,
            Typ       := Virtual(none),
            ExceptionCount := 0,
            Exceptions:= LAMBDA (x:ExcpIndex) : noExcpInfo,
            maxPC     := 6,
            Context   := 1,
            ByteCodes := LAMBDA (pc:PCOffset | pc<6): COND
              pc=0 -> aload(0),
```

```
          pc=1 -> bspush(2),
          pc=2 -> bspush(2),
          pc=3 -> invokevirtual(1,3),
          pc=4 -> putfield_s_this(0),
          pc=5 -> return,
          ELSE -> noInstr  ENDCOND
          #),
m=1 ->  (#
        nArgs      := 3,
        Args       := LAMBDA (x:nat) : Reference,
        Class      := 0,
        maxLV      := 0,
        Typ        := Virtual(none),
        ExceptionCount := 0,
        Exceptions:= LAMBDA (x:ExcpIndex) : noExcpInfo,
        maxPC      := 27,
        Context    := 1,
        ByteCodes := LAMBDA (pc:PCOffset | pc<27): COND
          % ack 0 n = n+1
          pc=0  -> bspush(0),
          pc=1  -> sload(1),
          pc=2  -> if_scmplt(6),
          pc=3  -> sinc(2,1),
          pc=4  -> sload(2),
          pc=5  -> sreturn,
          % ack (m+1) 0 = ack m 1
          pc=6  -> bspush(0),
          pc=7  -> sload(2),
          pc=8  -> if_scmplt(15),
          pc=9  -> sinc(1,-1),
          pc=10 -> aload(0),
          pc=11 -> sload(1),
          pc=12 -> bspush(1),
          pc=13 -> invokevirtual(1,3),
          pc=14 -> sreturn,
          % ack (m+1) (n+1) = ack m (ack (m+1) n)
          pc=15 -> sinc(2,-1),
          pc=16 -> aload(0),
          pc=17 -> sload(1),
          pc=18 -> sload(2),
          pc=19 -> invokevirtual(1,3),
          pc=20 -> sstore(2),
```

```
            pc=21 -> sinc(1,-1),
            pc=22 -> aload(0),
            pc=23 -> sload(1),
            pc=24 -> sload(2),
            pc=25 -> invokevirtual(1,3),
            pc=26 -> sreturn,
            ELSE-> noInstr  ENDCOND
          #),
    ELSE -> noMethod
    ENDCOND
    #)

Classes : ClassComponent = (#
  Size    := 1,
  Classes := LAMBDA (c:ClassIndex | c<1) : (#
    Typ          := Class,
    Context      := 1,
    MethodCount  := 2,
    MethodTable  := LAMBDA (n:MethodIndex|n<2): 1,
    SuperClass   := noSuperClass,
    FieldCount   := 1,
    FieldTable   := LAMBDA (n:FieldIndex | n<1) :
                    (# Token:=none,Typ:=Short #)
    #)
  #)

Ackermann : CAPFile = (#
  MC := Methods,
  CC := Classes,
  SC := noStaticComponent
  #)
```

A.4. PVS Formalization

```
BasicTypes : THEORY
%--
%-- This theory defines types that are used by throughout the machine
%--
BEGIN

%--
```

```
%-- types used by the machine
%--
InstrArg        : TYPE = int                % arguments of instructions without special type
LocVarIndex     : TYPE = nat                % index into local variable arrays
FieldIndex      : TYPE = nat                % index into the field array of an object
StaticIndex     : TYPE = nat                % index into staticmem to a static field
MethodIndex     : TYPE = nat                % index to a method definition
PCOffset        : TYPE = nat                % offset into the ByteCodes array of a method
ContextIndex    : TYPE = nat                % context id
ExcpIndex       : TYPE = nat                % index that addresses an exception handler
ClassIndex      : TYPE = nat                % index to a class definition
ExceptionId     : TYPE = ClassIndex

MCTypes         : TYPE = {Reference, Short}   % types of memory cells known by the machine
ArrayType       : TYPE = {Short,Boolean}      % types for arrays of primitive types

ModifierType    : TYPE = {none,protected}     % modifier for class members (objectfields & methods)


%-- Classes are represented by an Id number.
%-- Class Ids are used to address classes in the "ClassComponent" and identify classes
%-- Exceptions have no index into the "ClassComponent" but need an Id to be distinguishable.

%-- Standard classes which are supported by the machine need a constant id.
noSuperClass    : ClassIndex    = 65535  % for not existing superclasses (= 0xffff)
ObjectClass     : ClassIndex    = 65536  % class "Object"
ThrowableClass  : ClassIndex    = 65537  % class "Throwable"

%-- Standard exceptions which are supported by the machine have a constant Id
ArithmeticException             : ExceptionId = 65538
ArrayStoreException             : ExceptionId = 65539
ClassCastExeption               : ExceptionId = 65540
IndexOutOfBoundException        : ExceptionId = 65541
ArrayIndexOutOfBoundException   : ExceptionId = 65542
NullPointerException            : ExceptionId = 65543
SecurityException               : ExceptionId = 65544
NegativeArraySizeException      : ExceptionId = 65545


%-- List of instructions known by the machine + noInstr as undefined instruction.
%-- Every instruction gets a list of arguments with the specified type.
Instruction  : DATATYPE
BEGIN
    aload              (x1 : LocVarIndex)             : isaload
    athrow                                            : isathrow
    bspush             (x1 : InstrArg)                 : isbspush
    getfield_s_this    (x1 : LocVarIndex)             : isgetfield_s_this
    goto               (x1 : PCOffset)                : isgoto
    if_scmplt          (x1 : PCOffset)                : isif_scmplt
    invokevirtual      (m1 : MethodIndex, nargs:posnat): isinvokevirtual
    new                (x1 : ClassIndex)              : isnew
    newarray           (x1 : ArrayType)               : isnewarray
    noInstr                                           : isnoinstr
    putfield_s_this    (x1 : FieldIndex)              : isputfield_s_this
    return                                            : isreturn
    sand                                              : issand
    sconst             (x1 : InstrArg)                : issconst
    sinc               (x1 : LocVarIndex,x2:InstrArg)  : issinc
    sload              (x1 : LocVarIndex)             : issload
```

```
    sneg                                               : issneg
    sor                                                : issor
    sreturn                                            : issreturn
    sstore              (x1 : LocVarIndex)             : isstore
END Instruction


%-- The type for objects on the heap
%-- "ATyp" denotes arrays of primitive types with "Typ" = array fields type, "Count" = array size
%-- "CTyp" denotes class instances with "Class" = type of object (Class Id)
ObjectType : DATATYPE
BEGIN
    ATyp (Typ: ArrayType, Count: FieldIndex) : isArray
    CTyp (Class : ClassIndex)                : isClass
END ObjectType


%-- The type of a method in the "MethodComponent"
%-- "Virtual" denotes virtual methods with "token" = modifier for this method
%-- "CLinit" denotes class initialization methods
MethodType : DATATYPE
BEGIN
    Virtual (token: ModifierType) : isVirtual
    Cinit                         : isCinit
    CLinit                        : isCLinit
    bottom                        : bottom?
END MethodType


%--
%-- The type of a class definition in the "ClassComponent"
%--
ClassType : DATATYPE
BEGIN
    Class     : isClass
    Interface : isInterface
END ClassType


%-- dummy function for security check
secure(oc:ContextIndex, mc:ContextIndex) : bool = TRUE


END BasicTypes


%--
%-- "Rlift" is an extended "lift"-type which offers an additional "null"-component
%--
Rlift[T: TYPE] : DATATYPE
BEGIN
    bottom: bottom?
    null: null?
    up(down: T): up?
END Rlift


%-- Result is the result type of the execution of one instruction, as there are :
%-- "Norm" no error/exception has occurred the machine is still running
```

```
%-- "Halt_Norm" the machine has halted normally i.e. no exception/error
%-- "Exception" the instruction caused an exception
%-- "Halt:Exception" the machine halted due to an uncaught exception
%-- "Undef" the result of the execution is undefined
Result[State, MemCell : TYPE] : DATATYPE
BEGIN
    IMPORTING BasicTypes
    Norm           (NormState : State)                         : isNorm
    Halt_Norm      (Halt_NormState : State)                    : isHalt_Norm
    Exception      (ExcpState : State, ep : MemCell)           : isExcepition
    Halt_Exception (Halt_ExceptionState : State, ep : MemCell) : isHalt_Exception
    Undef          (UndefState : State)                        : isUndef
END Result



CAPFileDefinition : THEORY
%-- This theory defines the types of all components of the CAP file and the tyoe of the CAP file
%-- The definitions are straight forward from the CAP file definition in the
%-- Java Card 2.1. Virtual Machine Specification - chapter 6
BEGIN  IMPORTING BasicTypes


%---------------------------------------------------------------------------------------------------
%-- The METHOD COMPONENT defines all methods and exception handlers in the CAP file.
%-- -> Java Card 2.1. Virtual Machine Specification - chapter 6.9
%---------------------------------------------------------------------------------------------------

%--
%-- type for one exception info definition
%--
ExcpInfo : TYPE = [#
    Start    : PCOffset,       % start offset for the active range of the exception
    Length   : nat,            % length of active range of the exception
    Handler  : PCOffset,        % index into ByteCodes array to the code that handles the exception
    ExcpType : ExceptionId     % the exception type handled by this handler
    #]

%--
%-- the type for one method definition (method info)
%--
MethodInfo : TYPE = [#
    Class          : ClassIndex,                    % Id of the defining class
    Typ            : MethodType,                    % the method type (cinit|clinit|virtual)
    nArgs          : posnat,                        % number of arguments of the method
    Args           : [{x:nat | x<nArgs} -> MCTypes],% the types of the arguments
    maxLV          : LocVarIndex,                   % number of local variables in the method
    maxPC          : PCOffset,                      % number of instructions in the ByteCodes array
    ByteCodes      : [{x:PCOffset | x<maxPC} -> Instruction],   % the method code
    ExceptionCount : ExcpIndex,                                 % number of throwable exceptions
    Exceptions     : [{n:ExcpIndex | n<ExceptionCount} -> ExcpInfo],  % the exception handler infos
    Context        : ContextIndex                   % the package this method is defined in
    #]

%--
%-- the method component
%--
MethodComponent : TYPE = [#
    Size    : MethodIndex,             % number of methods defined in the "MethodComponent"
```

```
    Methods : [{n:MethodIndex | n<Size} -> MethodInfo]  % list of method definitions
    #]


%------------------------------------------------------------------------------------------------
%-- The class component defines all classes in the CAP file.
%-- -> Java Card 2.1. Virtual Machine Specification - chapter 6.8
%------------------------------------------------------------------------------------------------

%--
%-- info for one class field
%--
FieldInfo : TYPE = [#
    Token : ModifierType,  % the field modifier
    Typ   : MCTypes        % the field type
    #]

%--
%-- type for one class definition
%--
ClassInfo : TYPE = [#
    Typ         : ClassType,                    % type of the class (interface|class)
    SuperClass  : ClassIndex,                   % the Id of the superclass of this class
    FieldCount  : FieldIndex,                   % number of fields in this class (+ superclass)
    FieldTable  : [{n:FieldIndex | n<FieldCount}->FieldInfo], % modifiers for fields in this class
    MethodCount : MethodIndex,                  % number of methods in this class (+ superclass)
    MethodTable : [{n:MethodIndex | n<MethodCount}->MethodIndex],% Ids of the methods in this class
    Context     : ContextIndex                  % the package this class is defined in
    #]

%--
%-- the class component
%--
ClassComponent : TYPE = [#
    Size    : ClassIndex,                         % number of classes in the "ClassComponent"
    Classes : [{n:ClassIndex | n<Size} -> ClassInfo]  % list of class infos
    #]


%------------------------------------------------------------------------------------------------
%-- The STATIC FIELD COMPONENT defines all static fields in the CAP file.
%-- -> Java Card 2.1. Virtual Machine Specification - chapter 6.10
%------------------------------------------------------------------------------------------------

%--
%-- type for one static field
%--
StaticFieldInfo : TYPE = [#
    Modifier : ModifierType,  % the modifier for the static field
    Class    : ClassIndex     % the class this static field is defined in
    #]

%--
%-- the type for the static field component
%--
StaticComponent : TYPE = [#
    Size       : StaticIndex,           % the number of static fields in the "StaticComponent"
    StaticInfo : [{n:StaticIndex | n<Size} -> StaticFieldInfo]  % list of static field infos
    #]
```

```
%--
%-- Constants for not initialized CAP file parts
%--
noExcpInfo        : ExcpInfo
noMethod          : MethodInfo
noClassComponent  : ClassComponent
noClass           : ClassInfo
noFieldInfo       : FieldInfo
noStaticComponent : StaticComponent


%-- The "CAPFile" combines all components in a record.
%-- It represents the type of the CAP file.
%-- Right now only components that are needed by this formalization are included.
CAPFile : TYPE = [#
    MC : MethodComponent,
    CC : ClassComponent,
    SC : StaticComponent
    #]


%-------------------------------------------------------------------------------------------------
%-- helper functions for access to the CAP file components
%-------------------------------------------------------------------------------------------------


%-- check if "sub" is a subclass of "sup" (no check for circular class definitions yet)
isSubClass (sub, sup : ClassIndex, c : ClassComponent) : INDUCTIVE bool =
    IF    (sub>=noSuperClass) THEN FALSE
    ELSIF (sub=sup)           THEN TRUE
    ELSIF (sub<c`Size)        THEN isSubClass (c`Classes(sub)`SuperClass, sup, c)
                              ELSE FALSE
    ENDIF

%- returns the instructions addressed by "method" and "pc" or "bottom" if instruction doesn't exist
GetInstruction (m:MethodComponent, method:MethodIndex, pc:PCOffset) : lift[Instruction] =
    IF (method < m`Size AND pc < m`Methods(method)`maxPC)
    THEN up(m`Methods(method)`ByteCodes(pc)) ELSE bottom
    ENDIF

%-- returns the package id (context) of a method or "bottom" if the method doesn't exist
getMthdContext (x:MethodIndex, m:MethodComponent) : lift[ContextIndex] =
    IF (x<m`Size)
    THEN up(m`Methods(x)`Context) ELSE bottom
    ENDIF

%-- returns the exception info of an exception or "bottom" if the exceptions info doesn't exist
GetExcpetionInfo (x:ExcpIndex, m:MethodInfo) : lift[ExcpInfo] =
    IF x<m`ExceptionCount THEN up(m`Exceptions(x)) ELSE bottom ENDIF

%-- returns the method id of a method addressed by an index into the method table of a class
%-- or "bottom" if the method doesn't exist in the class
getClassMethod (x:MethodIndex,m:MethodComponent,i: ClassIndex,c:ClassComponent):lift[MethodIndex]=
    IF (i<c`Size) AND (x<c`Classes(i)`MethodCount)
    THEN up(c`Classes(i)`MethodTable(x)) ELSE bottom
    ENDIF
```

```
%-- returns the "MethodType" of "method" or "bottom" if the method/type doesn't exist
getMethodTyp (method:MethodIndex, m:MethodComponent) : MethodType =
    IF (method<m`Size) THEN m`Methods(method)`Typ ELSE bottom ENDIF

%-- checks if a static field exists in the "StaticComponent"
isStatic (x:StaticIndex,t:StaticComponent) : bool =
    IF (x < t`Size) THEN TRUE ELSE FALSE ENDIF


END CAPFileDefinition



MemCellOperations[MemCell : TYPE] : THEORY
%-- "MemCellOperations" defines prototypes for all operations on
%-- memory cells and the CAP file that MAY be checked by the machine
%-- The actual operations are defined in the theories that
%-- define "MemCell" i.e. "UntypedMemCell" & "TypedMemCell"
BEGIN
IMPORTING CAPFileDefinition, Rlift

%--
%-- the memory cell operations
%--
MemCellOps : TYPE =
    [#
    %-- the null reference
    Null           : MemCell,
    %-- compute a new reference
    NewReference   : [MemCell -> MemCell],
    %-- initialize memory cells
    initMemCell    : [MCTypes -> MemCell],
    %-- check type of memcell
    hasType        : [MemCell,MCTypes -> bool],
    %-- reference access
    isReference    : [MemCell ->bool],
    getReference   : [MemCell -> Rlift[MemCell]],
    %-- short access
    isShort        : [MemCell -> bool],
    toShort        : [int -> MemCell],
    ShortValue     : [MemCell -> lift[int]],
    %-- functions for local variables
    isLocVarIndex  : [LocVarIndex, MethodComponent, MethodIndex -> bool],
    getLV          : [LocVarIndex, MethodComponent, MethodIndex, MemCell -> lift[MemCell]],
    %-- functions for fields
    isFieldIndex   : [FieldIndex, ClassComponent, ObjectType -> bool],
    getOF          : [FieldIndex, ClassComponent, ObjectType, MemCell -> lift[MemCell]],
    accessableOF   : [FieldIndex, ClassComponent, ObjectType, MethodIndex, MethodComponent->bool],
    %-- functions for methods
    accessableMthd : [MethodIndex, ModifierType, MethodComponent, ClassIndex, MethodIndex,
                      ClassComponent -> bool],
    getPC          : [PCOffset, MethodComponent, MethodIndex -> lift[PCOffset]],
    %-- functions for classes
    getClass       : [ClassIndex, ClassComponent -> lift[ClassInfo]]
    #]

%--
%-- Constraints for properly defined MemCellOps
%--
```

```
MemCellConstraints (mco: MemCellOps) : bool =
    (mco`isShort(mco`initMemCell(Short))) AND
    (mco`getReference(mco`initMemCell(Reference)) /= bottom) AND
    (FORALL (x:int) : mco`isShort(mco`toShort(x))) AND
    (FORALL (x:MemCell) : mco`getReference(mco`NewReference(x)) /= bottom) AND
    (FORALL (x:MemCell) : mco`NewReference(x) /= mco`Null) AND
    (FORALL (x:MemCell) : (mco`getReference(x) = null) IFF (x=mco`Null))

END MemCellOperations



UntypedMemCell : THEORY
%-- This theory formalizes an untyped memory cell
%-- witch has no restrictions on types and range-
BEGIN

%-- UTMC = untyped memory cell
UTMC : TYPE = int
IMPORTING MemCellOperations[UTMC]


%-- the memory cell operations for the untyped memory cell
%-- all functions return true or up respectively
utops : MemCellOps[UTMC] =
    (#
    Null           := 0,

    NewReference   := LAMBDA (x:UTMC) : x+1,

    initMemCell    := LAMBDA (t:MCTypes) : 0,

    hasType        := LAMBDA (x:UTMC,t:MCTypes) : TRUE,

    isReference    := LAMBDA (x:UTMC) : TRUE,
    getReference   := LAMBDA (x:UTMC) : IF (x = 0) THEN null ELSE up(x) ENDIF,

    isShort        := LAMBDA (x:UTMC) : TRUE,
    toShort        := LAMBDA (x:int) : x,
    ShortValue        := LAMBDA (x:UTMC) : up(x),

    isLocVarIndex  := LAMBDA (x:LocVarIndex, m:MethodComponent, i:MethodIndex) : TRUE,
    getLV          := LAMBDA (x:LocVarIndex, m:MethodComponent, i:MethodIndex, mc:UTMC) : up(mc),

    isFieldIndex   := LAMBDA (x:FieldIndex, c:ClassComponent, t:ObjectType) : TRUE,
    getOF          := LAMBDA (x:FieldIndex, c:ClassComponent, t:ObjectType, mc:UTMC) : up(mc),
    accessableOF   := LAMBDA (x:FieldIndex, c:ClassComponent, t:ObjectType, i:MethodIndex,
                              m:MethodComponent) : TRUE,

    accessableMthd := LAMBDA (x:MethodIndex,
                              t:ModifierType,
                              m:MethodComponent,
                              class:ClassIndex,
                              i:MethodIndex, c:ClassComponent) : TRUE,

    getPC          := LAMBDA (x:PCOffset, m:MethodComponent, i:MethodIndex) : up(x),

    getClass       := LAMBDA (x:ClassIndex, c:ClassComponent) :
                      IF x<c`Size THEN up(c`Classes(x)) ELSE up(noClass) ENDIF
```

```
    #)

END UntypedMemCell



TypedMemCell : THEORY
%-- This theory formalizes a typed memory cell
%-- witch strongly typed but has no range restrictions.
BEGIN

%-- TMC = typed memory cell
TMC : DATATYPE
BEGIN
    Short (sValue : int) : isShort
    Ref (rValue : lift[nat]) : isRef
END TMC

IMPORTING MemCellOperations[TMC]


%-- the memory cell operations for the untyped memory cell
%-- all necessary type checks are performed
tops : MemCellOps[TMC] =
    (#
    Null           := Ref(bottom),

    NewReference   := LAMBDA (x:TMC) :
                      IF isRef(x) AND up?(rValue(x)) THEN Ref(up(down(rValue(x))+1))
                                   ELSE Ref(up(0)) ENDIF,

    initMemCell    := LAMBDA (t:MCTypes) : CASES t OF
                                       Reference : Ref(bottom),
                                       Short     : Short(0)
                                       ENDCASES,

    hasType        := LAMBDA (x:TMC,t:MCTypes) : CASES x OF
                                             Short(s) : (t = Short),
                                             Ref(r)   : (t = Reference)
                                                    ENDCASES,

    isReference    := LAMBDA (x:TMC) : isRef(x),
    getReference   := LAMBDA (x:TMC) : IF isRef(x) THEN
                                       IF bottom?(rValue(x)) THEN null
                                       ELSE up(x) ENDIF
                                   ELSE bottom ENDIF,

    isShort        := LAMBDA (x:TMC) : isShort(x),
    toShort        := LAMBDA (x:int) : Short(x),
    ShortValue     := LAMBDA (x:TMC) : IF isShort(x) THEN up(sValue(x)) ELSE bottom ENDIF,

    isLocVarIndex  := LAMBDA (x:LocVarIndex, m:MethodComponent, i:MethodIndex) :
                      (i < m`Size AND x < (m`Methods(i)`maxLV + m`Methods(i)`nArgs)),
    getLV          := LAMBDA (x:LocVarIndex, m:MethodComponent, i:MethodIndex, mc:TMC) :
                      IF (i < m`Size AND x < (m`Methods(i)`maxLV + m`Methods(i)`nArgs))
                      THEN up(mc) ELSE bottom ENDIF,

    isFieldIndex   := LAMBDA (x:FieldIndex, c:ClassComponent, i:ObjectType) :
                      CASES i OF
```

```
                CTyp(class)  : IF (class < c`Size AND x <= c`Classes(class)`FieldCount) THEN
                               TRUE ELSE FALSE ENDIF,
                ATyp(t,size) : IF x < size THEN TRUE ELSE FALSE ENDIF
                ENDCASES,

getOF          := LAMBDA (x:FieldIndex, c:ClassComponent, i:ObjectType, mc:TMC):
                % no modifier check
                CASES i OF
                ATyp(t,size) : IF x < size THEN up(mc) ELSE bottom ENDIF,
                CTyp(class)  : IF (class<c`Size
                               AND x <= c`Classes(class)`FieldCount) THEN
                                   up(mc) ELSE bottom ENDIF
                ENDCASES,

accessableOF   := LAMBDA (x:FieldIndex, c:ClassComponent, t:ObjectType, i:MethodIndex,
                        m:MethodComponent) :
                CASES t OF
                ATyp(tp,size) : TRUE,
                CTyp(class)   : IF (class<c`Size)
                                AND (x<c`Classes(class)`FieldCount) THEN
                                    IF (c`Classes(class)`FieldTable(x)`Token=protected) THEN
                                        IF  (i < m`Size)
                                        AND (m`Methods(i)`Class<c`Size)
                                        AND (x < c`Classes(m`Methods(i)`Class)`FieldCount)
                                        AND isSubClass (class,m`Methods(i)`Class,c) THEN TRUE
                                        ELSE FALSE ENDIF
                                    ELSE TRUE ENDIF
                                ELSE FALSE ENDIF
                ENDCASES,

accessableMthd := LAMBDA (method : MethodIndex,     % method to check
                        token : ModifierType,     % modifier of method to check
                        m:MethodComponent,
                        class:ClassIndex,         % object of method to check
                        cmethod : MethodIndex,    % current method
                        c:ClassComponent) :
                CASES token OF
                protected : IF  (method < m`Size)
                            AND (cmethod < m`Size)
                            % object class < current class
                            AND (m`Methods(cmethod)`Class < c`Size)
                            AND isSubClass (class,m`Methods(cmethod)`Class,c)
                            % current class < method defining class
                            AND (m`Methods(cmethod)`Class < c`Size)
                            AND isSubClass (m`Methods(cmethod)`Class,
                                            m`Methods(method)`Class,c) THEN
                                 TRUE
                            ELSE FALSE ENDIF,
                none : TRUE
                ENDCASES,

getPC          := LAMBDA (pc:PCOffset, m:MethodComponent, method:MethodIndex) :
                IF (method < m`Size)
                AND (pc < m`Methods(method)`maxPC) THEN up(pc)
                ELSE up(pc) ENDIF,

getClass       := LAMBDA (x:ClassIndex, c:ClassComponent) :
                IF x<c`Size THEN up(c`Classes(x)) ELSE bottom ENDIF
#)
```

```
END TypedMemCell


StateDefinition [
        MemCell : TYPE,
        (IMPORTING MemCellOperations[MemCell]) mcops : (MemCellConstraints[MemCell])
        ] : THEORY
% This theory formalizes the machine state which is partially described in chapter 3 of the
% Java Card Virtual Machine Specification.
% The machine state depends on a formalization of the memory cell above.
BEGIN

IMPORTING BasicTypes


%-- An object cell is a record used for storing all data of an object|array on the heap.
%-- The object cell has a field for
%-- "Typ" is the type of the object, either "ATyp" for arrays or "CTyp" for class instances
%-- "Field" is the list of fields of the object
%-- "Context" is the context this object is assigned to
ObjectCell : TYPE = [#
    Typ     : ObjectType,
    Field   : [FieldIndex -> MemCell],
    Context : ContextIndex
    #]


%-- Frame cell is the record which stores frames. A frame is an element of the stack memory,
%-- where all information of an executing method is stored.
%-- The frame cell has fields for
%-- "PC" is an index into the ByteCodes array to the instruction which has to be evaluated next.
%-- "CrntMthd" is the Id of the method the frame belongs to.
%-- "localVars" is the list of local variables
%-- "OpernadStack" is the opperand stack
%-- "Context" is the context this method is assigned to
FrameCell : TYPE = [#
    PC : PCOffset,
    CrntMthd       : MethodIndex,
    LocalVariables : [LocVarIndex -> MemCell],
    OperandStack   : list[MemCell],
    Class          : ClassIndex,
    Context        : ContextIndex
    #]


%-- "State" is the record which stores all state information of the machine during execution.
%-- "HeapTop" is the next free position on the heap memory
%-- "HeapMem" the heap memory is the list of run time objects referenced by a reference.
%-- "StackMem" the stack memory is the place where method frames are stored
%-- "StaticMem" the static memory is the place where all static variables are stored
State : TYPE = [#
    HeapTop   : MemCell,
    HeapMem   : [ MemCell -> ObjectCell],
    StackMem  : list[FrameCell],
    StaticMem : [StaticIndex -> MemCell]
    #]
```

```
%-- "Result" is imported here to have the same instantiation of it throughout the machine.
%-- It is needed by all theories which use "StateDefinition"
IMPORTING Result[State, MemCell]


%-- the type if the Java Card Virtual Machine
JCVMType : TYPE = [State, CAPFile -> Result[State, MemCell]]


%-- Constants for special or not initialized ObjectCell and FrameCell
%-- "noObject" not initialized ObjectCell
%-- "noFrame" not initializes / empty FrameCell
noObject : ObjectCell
noFrame : FrameCell = (#
    PC             := 0,
    Class          := noSuperClass,
    CrntMthd       := 0,
    LocalVariables := LAMBDA (x:nat) : Null(mcops),
    OperandStack   := null,
    Context        := 0
    #)


END StateDefinition




ExceptionHandling [
    MemCell : TYPE,
    (IMPORTING MemCellOperations[MemCell]) mcops : (MemCellConstraints[MemCell])
    ] : THEORY
%-- This theory defines the handling of exceptions thrown by the program or the machine.
%-- Exception handling starts when an exception is thrown and ranges to the point where
%-- a handler is found and the machine continues execution with the handling method. If
%-- no handling method was found then the machine halts.
%--
%-- Exception handling is defined in the Java Virtual Machine Specification (2.16)
BEGIN

IMPORTING StateDefinition[MemCell, mcops] , CAPFileDefinition


%-- "GetExcpHandler" searches all exception handler infos of the current method for an exception
%-- handler info that has the type of the thrown exception and whose range is active for the
%-- instruction, that caused the exception. The exception handler infos are sorted in a way, that
%-- the first found exception handler info which is applicable is always the right one.
%--
%-- This function returns an index into the exception handler info of the current method to the
%-- exception handler, which handles the exception or bottom if no applicable exception handler
%-- was found.
GetExcpHandler (exid:ExceptionId,
                method:MethodInfo,
                n:ExcpIndex,
                pc:PCOffset) : RECURSIVE lift[ExcpIndex] =
%-- fetch exception info (n)
CASES GetExcpetionInfo(n,method) OF
bottom : bottom,    %-- exception info not found -> handler not found
```

```
up(Handler) :       %-- exception info found
    %-- pc is below range or no more handler defined -> no handler found
    IF    (pc > Handler`Start + Handler`Length) THEN bottom
    %-- pc is ahead of range -> try next
    ELSIF pc < (Handler`Start) THEN  GetExcpHandler(exid,method,n+1,pc)
    %-- matching handler found
    ELSIF (Handler`ExcpType = exid) OR (Handler`ExcpType= 0) THEN up(n)
    %-- else try next
    ELSE  GetExcpHandler(exid, method, n+1, pc) ENDIF
ENDCASES
MEASURE n


%-- "SearchHandler" calls "GetExcpHandler" to find the handling method for a thrown exception.
%-- If no handling method was found, then the frame of the calling method is made current and
%-- "SearchHandler" is called again. If a handling method was found, then the current frame is
%-- discarded and a frame for the handling method is installed. Execution then continues in the
%-- handling method.
%-- If no handling method could be found an the main method is reached, then "bottom" is returned.
SearchHandler (excpref : MemCell,
               s : State,
               m : MethodComponent) : RECURSIVE lift[State] =
%-- check stackmem
CASES StackMem(s) OF
null : bottom, % no frame on stack -> should not happen
cons (crntFrame,restStack) :
    %-- compute exception id
    CASES s`HeapMem(excpref)`Typ OF
    CTyp(excpid) : IF (crntFrame`CrntMthd<m`Size) THEN
        CASES GetExcpHandler(0, m`Methods(crntFrame`CrntMthd), 0, crntFrame`PC) OF
        %-- we found the matching handler -> continue execution at handler code
        up(Handler) : up(s WITH [(StackMem) := cons(crntFrame WITH [
                                 PC := Handler,
                                 OperandStack := (: excpref :)
                                 ], restStack)
                            ]),
        %-- we found no matching handler in this method -> try in calling method
        bottom : CASES restStack OF
                 null : bottom,                    %-- no calling method -> stop
                 cons (crntFrame2,restStack2) :    %-- look for handler in method of frame
                     SearchHandler(excpref, s WITH [
                         (StackMem) := cons(crntFrame2 WITH [
                             OperandStack := cons(excpref, crntFrame2`OperandStack)
                             ], restStack2)
                     ],m)
                 ENDCASES
        ENDCASES ELSE bottom ENDIF %-- current method does not exist -> should not happen
    ELSE bottom
    ENDCASES
ENDCASES
MEASURE length(s`StackMem)


%-- "throw" creates an exception object of type "Excpid" on the heap and returns the result
%-- "Exception". This function is used for exceptions thrown by the machine.
throw (Excpid : ExceptionId, s : State) : Result[State, MemCell] =
Exception (s WITH [
                  (HeapTop) := NewReference(mcops)(s`HeapTop),
                  (HeapMem)(HeapTop(s))(Typ) := CTyp(Excpid)
```

```
                  ],s`HeapTop
          )


END ExceptionHandling



InstructionSemantics [
    MemCell : TYPE,
    (IMPORTING MemCellOperations[MemCell]) mcops : (MemCellConstraints[MemCell])
    ] : THEORY
%-- This theory contains the semantics functions for the instructions of the
%-- Java Card Virtual Machine.
%-- The semantics functions are the formalizations of the instructions which are described
%-- in the Java Card Virtual Machine Specification (chapter 7)
BEGIN

IMPORTING StateDefinition[MemCell, mcops], ExceptionHandling[MemCell, mcops]


%-- athrow
sem_athrow : JCVMType = LAMBDA (s:State, cap:CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef(s),
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef(s),
    cons (OpTop,restOp) :
        %-- check operand stack top to be reference
        CASES getReference(mcops)(OpTop) OF
        bottom : Undef(s),
        null : throw(NullPointerException, s),
        up(ref) :
            %-- fetch type of exception object
            CASES s`HeapMem(ref)`Typ OF
            ATyp(a,t) : Undef(s),
            CTyp(class) :
                %-- exception object must be subclass of throwable
                IF NOT(isSubClass(class, ThrowableClass, cap`CC)) THEN Undef(s)
                %-- check security
                ELSIF NOT(secure(crntFrame`Context,s`HeapMem(ref)`Context))
                    THEN throw(SecurityException,s)
%-- state update: return exception result
                ELSE Exception (s,ref)
                ENDIF
            ENDCASES
        ENDCASES
    ENDCASES
ENDCASES


%-- sload
sem_sload : [LocVarIndex -> JCVMType] = LAMBDA (Arg:nat) : LAMBDA (s:State, cap:CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef(s),
```

```
cons (crntFrame,restStack) :
    %-- fetch local variable Arg
    CASES getLV(mcops)(Arg, cap`MC, crntFrame`CrntMthd, crntFrame`LocalVariables(Arg)) OF
    bottom : Undef (s),
    up(MC) :
        IF NOT(isShort(mcops)(MC)) THEN Undef(s) %-- local variable Arg must be a short
        ELSE
%-- state update: push local variable Arg onto operand stack and increase PC
            Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                                    (OperandStack) := cons(MC, crntFrame`OperandStack),
                                    (PC) := crntFrame`PC + 1
                                    ], restStack )
                        ])
        ENDIF
    ENDCASES
ENDCASES


%-- aload
sem_aload : [LocVarIndex -> JCVMType] = LAMBDA (Arg:nat) : LAMBDA (s:State, cap:CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef(s),
cons (crntFrame,restStack) :
    %-- fetch local variable Arg
    CASES getLV(mcops)(Arg, cap`MC, crntFrame`CrntMthd, crntFrame`LocalVariables(Arg)) OF
    bottom : Undef (s),
    up(MC) :
        IF NOT(isReference(mcops)(MC)) THEN Undef(s) %-- local variable Arg must be a reference
        ELSE
%-- state update: push local variable Arg onto operand stack and increase PC
            Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                                    (OperandStack) := cons(MC, crntFrame`OperandStack),
                                    (PC) := crntFrame`PC + 1
                                    ], restStack )
                        ])
        ENDIF
    ENDCASES
ENDCASES


%-- getfield_s_this
sem_getfield_s_this : [FieldIndex -> JCVMType] =
LAMBDA (Arg : FieldIndex) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- check if local variable 0 is a reference
    CASES getReference(mcops)(crntFrame`LocalVariables(0)) OF
    bottom : Undef(s),
    null : throw (NullPointerException, s),
    up(ref) :
        %-- check security of operation
        IF NOT(secure(crntFrame`Context,s`HeapMem(ref)`Context))
            THEN throw(SecurityException,s)
        %-- check modifiers of the object field
        ELSIF NOT(accessableOF(mcops)(Arg, cap`CC, s`HeapMem(ref)`Typ, crntFrame`CrntMthd, cap`MC))
            THEN Undef(s)
```

```
        ELSE
            %-- fetch the object field
            CASES getOF(mcops)(Arg, cap`CC, s`HeapMem(ref)`Typ, s`HeapMem(ref)`Field(Arg) ) OF
            bottom : Undef(s),
            up(MC) :
%-- state update: push object field onto operant stack, increase PC
                Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                         (OperandStack) := cons(MC, crntFrame`OperandStack),
                         (PC) := crntFrame`PC + 1
                         ] , restStack)
                    ])
            ENDCASES
        ENDIF
    ENDCASES
ENDCASES


%-- putfield_s_this
sem_putfield_s_this : [FieldIndex -> JCVMType] =
LAMBDA (Arg : FieldIndex) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- check if local variable 0 is a reference
    CASES getReference(mcops)(crntFrame`LocalVariables(0)) OF
    bottom : Undef (s),
    null : throw (NullPointerException, s),
    up(ref) :
        %-- check security of operation
        IF NOT(secure(crntFrame`Context,s`HeapMem(ref)`Context))
            THEN throw(SecurityException,s)
        %-- check modifiers of object field
        ELSIF NOT(accessableOF(mcops)(Arg, cap`CC, s`HeapMem(ref)`Typ, crntFrame`CrntMthd, cap`MC))
            THEN Undef(s)
        ELSE
            % fetch operand stack top
            CASES crntFrame`OperandStack OF
            null : Undef(s),
            cons (OpTop, restOp) :
                %-- check if field exists
                IF isFieldIndex(mcops)(Arg, cap`CC, s`HeapMem(ref)`Typ)
                %-- check is field is of type short
                AND isShort(mcops)(s`HeapMem(ref)`Field(Arg)) THEN
% state update : pop operand stack top, set object filed to operand-stack-top, increase PC
                    Norm(s WITH [(HeapMem)(ref)`Field(Arg) := OpTop,
                                 (StackMem):= cons( crntFrame WITH [
                                     (OperandStack) := restOp,
                                     (PC) := crntFrame`PC + 1
                                     ], restStack)
                                 ])
                ELSE Undef(s) ENDIF
            ENDCASES
        ENDIF
    ENDCASES
ENDCASES


%-- sneg
```

```
sem_sneg : JCVMType = LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    % fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop, restOp) :
        % get value of operand stack top
        CASES ShortValue(mcops)(OpTop) OF
        bottom : Undef (s),
        up (svalue) :
%-- state update: push negated operand stack top onto operand stack, increase PC
            Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                            (OperandStack) := cons(toShort(mcops)(1-svalue),restOp),
                            (PC) := (crntFrame`PC +1)
                            ], restStack)
                        ])
        ENDCASES
    ENDCASES
ENDCASES


%-- sand
sem_sand : JCVMType = LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop, restOp) :
        %-- compute value of operand stack top
        CASES ShortValue(mcops)(OpTop) OF
        bottom : Undef (s),
        up (svalue1) :
            %-- fetch next operand stack top
            CASES restOp OF
            null : Undef (s),
            cons (OpTop2,restOp2) :
                %-- compute value of next operand stack top
                CASES ShortValue(mcops)(OpTop2) OF
                bottom : Undef (s),
                up (svalue2) :
%-- state update: push result onto operand stack. increase PC
                    Norm(s WITH [
                       (StackMem) := cons(crntFrame WITH [
                           (OperandStack) := cons(toShort(mcops)(
                                                 IF svalue1 = 0 THEN 0 ELSE svalue2 ENDIF),restOp2),
                           (PC) := (crntFrame`PC +1)
                           ], restStack)
                       ])
                ENDCASES
            ENDCASES
        ENDCASES
    ENDCASES
ENDCASES
```

```
%-- sor
sem_sor : JCVMType = LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop, restOp) :
        %-- compute value of operand stack top
        CASES ShortValue(mcops)(OpTop) OF
        bottom : Undef (s),
        up (svalue1) :
            %-- fetch next operand stack top
            CASES restOp OF
            null : Undef (s),
            cons (OpTop2,restOp2) :
                %-- compute value of next operand stack rest top
                CASES ShortValue(mcops)(OpTop2) OF
                bottom : Undef (s),
                up (svalue2) :
%-- state update: push result onto operand stack, increase PC
                    Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                        (OperandStack) :=
                            cons(toShort(mcops)(
                                     IF svalue1 /= 0 THEN 1 ELSE svalue2 ENDIF), restOp2),
                        (PC) := crntFrame`PC +1
                        ], restStack)
                    ])
                ENDCASES
            ENDCASES
        ENDCASES
    ENDCASES
ENDCASES


%-- sconst
sem_sconst : [InstrArg -> JCVMType] = LAMBDA (Arg : InstrArg) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
%-- state update: push Arg onto operand stack, increase PC
    Norm(s WITH [(StackMem) := cons(crntFrame WITH [
        (OperandStack) := cons(toShort(mcops)(Arg), crntFrame`OperandStack ),
        (PC) := crntFrame`PC + 1
        ], restStack)
    ])
ENDCASES


%-- sstore
sem_sstore : [LocVarIndex -> JCVMType] = LAMBDA (Arg:nat) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
```

```
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop, restOp) :
        %-- check if Arg is a valid index into the local variable array of the method
        IF NOT(isLocVarIndex(mcops)(Arg, cap`MC, crntFrame`CrntMthd))
        %-- check if operand stack top is of type short
        OR NOT(isShort(mcops)(OpTop)) THEN Undef(s)
        ELSE
%-- state update: set local variable Arg to operand stack top,pop top from operand stack,increase PC
            Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                            (LocalVariables)(Arg) := OpTop,
                            (OperandStack) := restOp,
                            (PC) := crntFrame`PC +1
                            ], restStack)
                        ])
        ENDIF
    ENDCASES
ENDCASES


%-- astore
sem_astore : [LocVarIndex -> JCVMType] =
LAMBDA (Arg : LocVarIndex) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    % fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop, restOp) :
        %-- check if Arg is a valid index into the local variable array of the method
        IF NOT(isLocVarIndex(mcops)(Arg, cap`MC, crntFrame`CrntMthd))
        %-- check if operand stack top is of type short
        OR NOT(isReference(mcops)(OpTop)) THEN Undef(s)
        ELSE
%-- state update: set local variable Arg to operand stack top,pop top from operand stack,increase PC
            Norm(s WITH [(StackMem) := cons(crntFrame WITH [
                            (LocalVariables)(Arg) := OpTop,
                            (OperandStack) := restOp,
                            (PC) := crntFrame`PC +1
                            ], restStack)
                        ])
        ENDIF
    ENDCASES
ENDCASES


%-- goto
sem_goto : [PCOffset -> JCVMType] = LAMBDA (Arg : PCOffset) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- check if Arg is a valid PC
    CASES getPC(mcops)(Arg, cap`MC ,crntFrame`CrntMthd) OF
    bottom : Undef (s),
```

```
    up(PC) :
%-- state update: set PC to Arg
        Norm(s WITH [(StackMem):=cons(crntFrame WITH [(PC) := PC], restStack)])
    ENDCASES
ENDCASES


%-- sinc
sem_sinc : [LocVarIndex, InstrArg -> JCVMType] =
LAMBDA (Arg1 : LocVarIndex, Arg2 : InstrArg) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch local variable Arg1
    CASES getLV(mcops)(Arg1, cap`MC, crntFrame`CrntMthd, crntFrame`LocalVariables(Arg1)) OF
    bottom : Undef (s),
    up(MC) :
        %-- compute value of local variable Arg1
        CASES ShortValue(mcops)(MC) OF
        bottom : Undef (s),
        up(svalue) :
%-- state update: increase local variable Arg1, increase PC
            Norm(s WITH [(StackMem):= cons(crntFrame WITH [
                            (LocalVariables)(Arg1) := toShort(mcops)(svalue + Arg2),
                            (PC) := crntFrame`PC +1
                            ], restStack)
                        ])
        ENDCASES
    ENDCASES
ENDCASES


%-- bspush
sem_bspush : [InstrArg -> JCVMType] = LAMBDA (Arg : InstrArg) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
%-- state update: push Arg onto operand stack, increase PC
    Norm(s WITH [(StackMem):= cons(crntFrame WITH [
              (OperandStack):= cons(toShort(mcops)(Arg),crntFrame`OperandStack),
              (PC) := crntFrame`PC +1
              ], restStack)
        ])
ENDCASES


%-- if_scmplt
sem_if_scmplt : [PCOffset -> JCVMType] =
LAMBDA (Arg : PCOffset) : LAMBDA (s : State, cap : CAPFile) :
% -- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop, restOp) :
```

```
        %-- fetch next operand stack top
        CASES restOp OF
        null : Undef (s),
        cons (OpTop2,restOp2) :
            %-- compute value of operand stack top
            CASES ShortValue(mcops)(OpTop) OF
            bottom : Undef (s),
            up(svalue1) :
            %-- compute value of next operand stack top
                CASES ShortValue(mcops)(OpTop2) OF
                bottom : Undef (s),
                up(svalue2) :
                    %-- compare operand stack top and next operand stack top
                    IF svalue2 < svalue1 THEN
                        %-- check if Arg is a valid PC
                        CASES getPC(mcops)(Arg, cap`MC, crntFrame`CrntMthd) OF
                        bottom : Undef (s),
                        up(PC) :
%-- state update: pop 2 topmost operand stack entries, set PC to Arg
                            Norm(s WITH [(StackMem):= cons(crntFrame WITH [
                                    (OperandStack):=restOp2,
                                    (PC):= Arg
                                    ], restStack)
                                ])
                        ENDCASES
                    ELSE
%-- state update: pop 2 topmost operand stack entries, increase PC
                        Norm(s WITH [(StackMem):= cons(crntFrame WITH [
                                    (OperandStack):=restOp2,
                                    (PC):= crntFrame`PC +1
                                    ], restStack)
                            ])
                    ENDIF
                ENDCASES
            ENDCASES
         ENDCASES
     ENDCASES
ENDCASES


%-- "PushArgs" pop arguments from operand stack of current frame and push them into local variables
%-- of new frame
%-- Arguments: the new frame (empty), the current frame (with arguments on the operand
%--            stack, number of arguments
%-- Result: tuple of 2 frames (new frame, current frame) or bottom
PushArgs(newFrame, crntFrame : FrameCell, n : posnat) : RECURSIVE lift[[FrameCell,FrameCell]] =
IF n=0 THEN up((newFrame,crntFrame)) %-- finished
ELSE
    %-- fetch operand stack top from current frame
    CASES crntFrame`OperandStack OF
    null : bottom,
    cons(OpTop, restOp) :
        % pop operand stack top from current frame and push it into local variable (n) of new frame
        PushArgs(newFrame WITH [(LocalVariables)(n-1) := OpTop ],
                 crntFrame WITH [(OperandStack) := restOp],
                 n-1)
        ENDCASES
    ENDIF
MEASURE n
```

```
%-- invokevirtual
sem_invokevirtual : [MethodIndex, posnat -> JCVMType] =
LAMBDA (Arg : MethodIndex, nargs : posnat) : LAMBDA (s : State, cap : CAPFile) :
% -- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %- pop arguments from operand stack of current frame and push into local variables of new frame
    CASES PushArgs(noFrame,crntFrame, nargs) OF
    bottom : Undef (s),
    up(Frames) :
        %-- local variable of new frame must contain a reference to the owning object
        CASES getReference(mcops)(Frames`1`LocalVariables(0)) OF
        bottom : Undef (s),
        null : throw(NullPointerException, s),
        up(ref) :
            %-- check type of owning object
            CASES s`HeapMem(ref)`Typ OF
            ATyp(tp,size) : Undef (s), %-- class Object must be used here, which doesn't exist jet
            CTyp(class):
                %-- compute method Id
                CASES getClassMethod(Arg, cap`MC, class, cap`CC) OF
                bottom : Undef (s),
                up(method) :
                    %-- check type of method
                    CASES getMethodTyp(method, cap`MC) OF
                    bottom: Undef (s),
                    Cinit : Undef (s),
                    CLinit : Undef (s),
                    Virtual(token) :
                        %-- check accessibility of method
                        IF NOT(accessableMthd(mcops)(method,token,cap`MC,class,Arg,cap`CC))
                            THEN Undef(s)
                        ELSE
                            %-- compute context of method
                            CASES getMthdContext(method,cap`MC) OF
                            bottom : Undef (s),
                            up (context) :
                                %-- check security of operation
                                IF NOT(secure(crntFrame`Context, context)) THEN
                                    throw (SecurityException, s)
                                ELSE
%-- state update: push new frame onto stackmem + set CrntMthd, PC, OperandStack, Context ;
%--              increase PC of invoking method
                                    Norm(s WITH [(StackMem) := cons(Frames`1 WITH [
                                            (CrntMthd) := method,
                                            (PC) := 0,
                                            (OperandStack) := null,
                                            (Context) := context
                                            ], cons(Frames`2 WITH [
                                                (PC) := Frames`2`PC +1],restStack))
                                        ])
                                ENDIF
                            ENDCASES
                        ENDIF
                    ENDCASES
                ENDCASES
```

```
            ENDCASES
        ENDCASES
    ENDCASES
ENDCASES


%-- return
sem_return : JCVMType = LAMBDA (s : State, cap : CAPFile) :
% -- fetch current frame
CASES s`StackMem OF
null : Undef (s), % stackmem empty should not happen
cons (crntFrame,restStack) :
%-- state update: pop current frame from stackmem
    Norm(s WITH [(StackMem) := restStack])
ENDCASES


%-- sreturn
sem_sreturn : JCVMType = LAMBDA (s : State, cap : CAPFile) :
% -- fetch current frame
CASES s`StackMem OF
null : Undef (s), % should not happen
cons (crntFrame,restStack) :
    % -- fetch next current frame
    CASES restStack OF
    null : Undef (s),
    cons (crntFrame2,restStack2) :
        %-- fetch operand stack top of current frame
        CASES crntFrame`OperandStack OF
        null : Undef (s),
        cons (OpTop,restOp) :
            %-- operand stack top of current frame must be of type short
            IF not(isShort(mcops)(OpTop)) THEN Undef (s)
            ELSE
%-- state update: pop current frame from stackmem and push operand stack top of current frame
%--              onto operand stack of next current frame
                Norm(s WITH [(StackMem) := cons(crntFrame2 WITH [
                    (OperandStack) := cons(OpTop, crntFrame2`OperandStack)
                    ] ,restStack2)
                ])
            ENDIF
        ENDCASES
    ENDCASES
ENDCASES


%-- "initFields" set the fields of the new class instance to their initial values
%-- arguments: 0, classinfo of the new class instance
%-- returns: object fields array of the new class instance
initFields (n : nat, cinfo : ClassInfo) : RECURSIVE [nat -> MemCell] =
IF n<cinfo`FieldCount THEN
    initFields (n+1,cinfo) WITH [(n) := initMemCell(mcops)(cinfo`FieldTable(n)`Typ)]
ELSE LAMBDA (n:FieldIndex) : Null(mcops) ENDIF
MEASURE n


%-- new
sem_new : [ClassIndex -> JCVMType] = LAMBDA (Arg : ClassIndex) : LAMBDA (s : State, cap : CAPFile):
%-- fetch current frame
```

```
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch classinfo of the new class instance
    CASES getClass(mcops)(Arg, cap`CC) OF
    bottom    : Undef(s),
    up(cinfo) :
        %-- classinfo  must not be an interface definition
        CASES cinfo`Typ OF
        Interface : Undef(s),
        Class :
%-- state update: make new object; compute new heaptop;push new object reference onto operand stack
            Norm (s WITH [(HeapMem)(s`HeapTop) := (#
                                              Typ := CTyp(Arg),
                                              Field := initFields (0,cinfo),
                                              Context := crntFrame`CrntMthd #),
                          (HeapTop) := NewReference(mcops)(s`HeapTop),
                          (StackMem) := cons(crntFrame WITH [
                              (OperandStack) := cons(s`HeapTop,crntFrame`OperandStack),
                              (PC):= crntFrame`PC+1
                              ],restStack)
                          ])
        ENDCASES
    ENDCASES
ENDCASES


%-- newarray
sem_newarray : [ArrayType -> JCVMType] =
LAMBDA (Arg : ArrayType) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop,restOp) :
        %-- compute value of operand stack top = size of array
        CASES ShortValue(mcops)(OpTop) OF
        bottom : Undef(s),
        up(count) :
            %-- array size must be greater 0
            IF (count >= 0) THEN
                %-- fetch current context
                CASES getMthdContext(crntFrame`CrntMthd, cap`MC) OF
                bottom : Undef (s), % no current context should not happen
                up(context) :
%-- state update: install new array object; compute new heaptop;
%--              push new object reference onto operand top
                  Norm (s WITH [(HeapMem)(s`HeapTop) := (#
                                  Typ := ATyp(Arg,count),
                                  Field := LAMBDA (x:FieldIndex): Null(mcops),
                                  Context := context #),
                              (HeapTop) := NewReference(mcops)(s`HeapTop),
                              (StackMem) := cons(crntFrame WITH [
                                  (OperandStack) := cons(s`HeapTop,restOp),
                                  (PC) := crntFrame`PC+1],restStack)
                         ])
```

```
                ENDCASES
            ELSE  throw(NegativeArraySizeException, s)
            ENDIF
        ENDCASES
    ENDCASES
ENDCASES


%-- putstatic_s
sem_putstatic_s : [StaticIndex -> JCVMType] =
LAMBDA (Arg : StaticIndex) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- fetch operand stack top
    CASES crntFrame`OperandStack OF
    null : Undef (s),
    cons (OpTop,restOp) :
        %-- Arg must be index to a static field of type short
        IF (NOT(isStatic(Arg, cap`SC) OR NOT(isShort(mcops)(OpTop)))) THEN Undef(s)
        ELSE
%-- state update: set static field Arg to operand stack top, pop operand stack top, increase PC
            Norm(s WITH [(StaticMem) := s`StaticMem WITH [(Arg) := OpTop],
                         (StackMem) := cons(crntFrame WITH [(PC):= crntFrame`PC +1,
                                                   (OperandStack):= restOp],restStack)
                        ])
        ENDIF
    ENDCASES
ENDCASES


%-- jsr
sem_jsr : [InstrArg -> JCVMType] = LAMBDA (Arg : InstrArg) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
    %-- Arg must be a valid PC from the current PC
    CASES getPC(mcops)(crntFrame`PC+Arg, cap`MC, crntFrame`CrntMthd) OF
    bottom : Undef(s),
    up(newpc) :
%-- state update: push (PC+1) onto operand stack, set PC to new PC
        Norm(s WITH [(StackMem) := cons(crntFrame WITH [
            (OperandStack) := cons(toShort(mcops)(crntFrame`PC+1),crntFrame`OperandStack),
            (PC) := newpc
            ], restStack)
        ])
    ENDCASES
ENDCASES


%-- ret
sem_ret : [LocVarIndex -> JCVMType] =
LAMBDA (Arg : LocVarIndex) : LAMBDA (s : State, cap : CAPFile) :
%-- fetch current frame
CASES s`StackMem OF
null : Undef (s),
cons (crntFrame,restStack) :
```

```
    %-- Arg must be a valid index into the local variable array of the current frame
    IF NOT(isLocVarIndex(mcops)(Arg, cap`MC, crntFrame`CrntMthd)) THEN Undef(s) ELSE
        %-- compute value of local variable Arg
        CASES ShortValue(mcops)(crntFrame`LocalVariables(Arg)) OF
        bottom : Undef(s),
        up(newpc) :
%-- state update: set PC to the value in local variable Arg
            Norm(s WITH [(StackMem) := cons(crntFrame WITH [(PC):= newpc],restStack)])
        ENDCASES
    ENDIF
ENDCASES


END InstructionSemantics



JCVMSemantics
    [
    MemCell : TYPE,
    (IMPORTING MemCellOperations[MemCell]) mcops : (MemCellConstraints[MemCell])
    ] : THEORY
%-- semantics functions for the JCVM
BEGIN

IMPORTING InstructionSemantics[MemCell, mcops]


%--
%-- calls the semantics function for an instruction
%--
Semantics : [Instruction -> JCVMType] = LAMBDA (instr:Instruction) : LAMBDA (s:State, cap:CAPFile):
  CASES instr OF
    aload(x1)            : sem_aload           (x1)    (s, cap),
    athrow               : sem_athrow                  (s, cap),
    bspush(x1)           : sem_bspush          (x1)    (s, cap),
    getfield_s_this(x1)  : sem_getfield_s_this (x1)    (s, cap),
    goto(x1)             : sem_goto            (x1)    (s, cap),
    if_scmplt(x1)        : sem_if_scmplt       (x1)    (s, cap),
    invokevirtual(x1,x2) : sem_invokevirtual   (x1,x2) (s, cap),
    new(x1)              : sem_new             (x1)    (s, cap),
    newarray(x1)         : sem_newarray        (x1)    (s, cap),
    putfield_s_this(x1)  : sem_putfield_s_this (x1)    (s, cap),
    return               : sem_return                  (s, cap),
    sand                 : sem_sand                    (s, cap),
    sconst(x1)           : sem_sconst          (x1)    (s, cap),
    sinc(x1,x2)          : sem_sinc            (x1,x2) (s, cap),
    sload(x1)            : sem_sload           (x1)    (s, cap),
    sneg                 : sem_sneg                    (s, cap),
    sor                  : sem_sor                     (s, cap),
    sreturn              : sem_sreturn                 (s, cap),
    sstore(x1)           : sem_sstore          (x1)    (s, cap),
    noInstr              : Undef(s)
  ENDCASES


%--
%-- A constant for the "State"-type
%--
```

```
NullState : State = (#
    HeapTop    := Null(mcops),
    HeapMem    := LAMBDA (x:MemCell) : noObject,
    StackMem   := null,
    StaticMem  := LAMBDA (x:StaticIndex) : Null(mcops)
    #)


%--
%-- "InitState" takes a state and sets the state fields to reasonable values for execution start.
%-- a dummy function since applet initialization is not formalized
%--
InitState : [State -> State] = LAMBDA (s:State) :
s WITH [(StackMem) := cons( noFrame WITH [
                            (LocalVariables)(0) := NewReference(mcops)(Null(mcops))], StackMem(s)),
        (HeapTop) := NewReference(mcops)(NewReference(mcops)(Null(mcops))),
        `HeapMem(NewReference(mcops)(Null(mcops)))`Typ := CTyp(0),
        `HeapMem(NewReference(mcops)(Null(mcops)))`Context := 1
       ]


%--
%-- "isInitialized" is a predicate which checks if a state is initialized.
%-- can be used instead of "InitState"
%--
isInitialized : [State -> bool] = LAMBDA (s:State) :
      s`StackMem = cons(noFrame WITH [
                         (LocalVariables)(0) := NewReference(mcops)(Null(mcops))],null)
  AND s`HeapTop = NewReference(mcops)(NewReference(mcops)(Null(mcops)))
  AND s`HeapMem(NewReference(mcops)(Null(mcops)))`Typ = CTyp(0)
  AND s`HeapMem(NewReference(mcops)(Null(mcops)))`Context=1


%-- "Run" defines the machine run time iteration. It evaluates the results of instructions.
%-- If the result is "Norm" then it fetches the next instruction from the method component
%-- and calls "Run" with the result of the execution of the instruction with the specified
%-- semantics function.
%-- If the result is "Halt_Norm" then it returns "TRUE".
%-- If the result is "Exception" then the exception handling is performed
%-- "Run" can be seen as a predicate that is true iff the program terminates normally which
%-- is proven by executing the program.
Run (s : State, CAP : CAPFile) : INDUCTIVE bool =
%-- fetch current frame
  CASES s`StackMem OF
  null : TRUE,                  %-- empty stack -> computation ends successfully
  cons (crntFrame,restStack) :  %-- nonempty stack
      %-- fetch next instruction
      CASES GetInstruction(CAP`MC,crntFrame`CrntMthd,crntFrame`PC) OF
      bottom : FALSE, %-- next instruction does not exist, should not happen
      up(I) : %-- compute successor state
          CASES Semantics(I)( s, CAP) OF
          Norm(suc) : Run (suc, CAP),   %-- normal result -> run
          Exception(s,e) : %-- exception was thrown -> exception handling
              %-- compute exception handler
              CASES SearchHandler(e,s,CAP`MC) OF
              bottom : false,
              up(es) : Run (es, CAP)
              ENDCASES
          ELSE false
```

```
          ENDCASES
      ENDCASES
  ENDCASES


%--
%-- the state transition relation of one step execution
%--
Arrow (s1 : State, p:CAPFile, s2 : State) : bool =
    cons?(s1`StackMem) AND
    (up?(i) AND (Semantics(down(i))(s1,p) = Norm(s2))
    ) WHERE i = GetInstruction(p`MC,car(s1`StackMem)`CrntMthd,car(s1`StackMem)`PC)


%--
%-- the state transition relation in arbitrary steps
%--
ArrowStar (s1 : State, p:CAPFile, s2 : State) : INDUCTIVE bool =
    s1 = s2 OR EXISTS (s3 : State): Arrow (s1,p,s3) AND ArrowStar (s3,p,s2)


END JCVMSemantics
```

B. Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbst angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet, sowie Zitate kenntlich gemacht habe.

Dresden, den 21. Februar 2001

Frank Felfe

C. Bibliography

[1] J. Gosling, B. Joy, G. L. Steele: *The Java Language Specification* Addison-Wesley, ISBN 0-201-63451-1

[2] T. Lindholm, F. Yellin: *The Java Virtual Machine Specification*second edition, Addison-Wesey, ISBN 0-201-43294-3

[3] SUN Microsystems Inc.: *Java Card 2.1.1 Virtual Machine Specification* http://java.sun.com

[4] SUN Microsystems Inc.: *Java Card 2.1.1. Runtime Environment Specification* http://java.sun.com

[5] SUN Microsystems Inc.: *Java Card 2.1.1. Application Interface Specification* http://java.sun.com

[6] C. Pusch: *Formalizing the Java Virtual Machine in Isabelle/HOL* http://www.in.tum.de/ pusch/

[7] C. Pusch: *Proving the soundness of a Java Bytecode Verifier Specification in Isabelle/HOL* http://www.in.tum.de/ pusch/

[8] Z. Qian: *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines* in Jim Alves-Foss, editor, Formal Syntax and Semantics of Java, volume 1523, of Lecture Notes in Computer Science

[9] G.Barthe, G.Dufay, L.Jakubiec, B.Serpette, S.Sousa, S.Yu *Formalization in Coq of the Java Card Virtual Machine* in 2nd ECOOP Workshop on Formal Techniques for Java Programs, Sophia Antipolis - France, 12 June 200

[10] J. v. d. Berg, M. Huisman, B. Jacobs, E. Poll *A Type-Theoretic Memory Model for Verification of Sequential Java Programs*

[11] M.Huisman *Reasoning about Java Programs in Higher-Order Logic, using* PVS *and Isabelle/HOL* PhD theses, Univ. Nijmegen, Dept. of Computer Science

[12] B. Jacobs *A Formalization of Java's Exception Mechanism* Technical Report CSI-R0015, October 2000, Univ. Nijmegen, Dept. of Computer Science

[13] The LOOP project, http://www.cs.kun.nl/ bart/LOOP

[14] B. Jacobs, J. Rutten *A Tutorial on (Co)Algebras and (Co)Induction* EATCS Bulletin, 62:222-259, 1997

[15] H. Reichel *An approach to object semantics based on terminal coalgebras* Math. Struct. Comp. Sci., 5:129-152, 1995

[16] J. Rothe, H. Tews, B. Jacobs *The Coalgebraic Class Specification Language CCSL* In 4th workshop on: Tools for System Design and Verification 2000

[17] E. Denny and T. Jensen *Correctness of the Java Method Lookup via Logical Relations*, in Gert Smolka, editor, Programming Languages and Systems, volume 1782, of Lecture Notes in Computer Science

[18] J. Bergstra, M. Loots *Empirial Semantics for Object-Oriented Programs*1999, Onderwijsinstituut CKI - Utrecht University, ISBN 90-393-2181-7

[19] *The Prototype Verification System* http://pvs.csl.sri.com

[20] E. Poll, J.v.d. Berg, B. Jacobs *Specification of the Java Card API in JML* Univ. Nijmegen, Dept of. Computer Science

[21] T. Nipkow, D. Oheimb *Machine Checking the Java Specification: Proving Type-Savety - Definitely* TU Munich, Dept. of Computer Science

[22] I. Attali, C. Caromel, M. Russo *A Formal Executable Semantics For Java*http://www.inria.fr/croap/java

[23] P.Bertelsen *Dynamic Semantics of Java Byte-code* Univ. of Copenhagen, pmb@dina.kvl.dk

[24] R. M. Cohen *The defensive Java Virtual Machine specification* Technical report, Computational Logic Inc., 1997 Draft version

[25] G.T. Leavens, A.L. Baker, and C. Ruby. *Preliminary design of JML: A behavioral interface specification language for Java* Technical report, Dept. of Comp. Sci., Iowa State Univ., 1999. Tech. Rep. 98-06

Index