Flow Logic for Carmel

Authors	: René Rydhof Hansen
Date	: June 28, 2002
Number	: SECSAFE-IMM-001-1.5
Classification	: Public

1 Introduction

The aim of this paper is to outline a framework for rapidly creating analyses of the Java Card Virtual Machine Language (JCVML). This is illustrated by the development of a simple control flow analysis for a subset of JCVML, called Carmel. The work was primarily motivated by the work of Freund and Mitchell on formalising the Java Virtual Machine Language (JVML) bytecode verifier as a typesystem, cf. [3], and a primary goal for the present work has been to illustrate how certain typesystems can be rewritten as Flow Logic specifications, while retaining the succinctness and elegance of the typesystem, and to enhance the Flow Logic notation where needed.

The rest of the paper is organised in the following way. Section 2 describes and discusses the subset of JCVML/Carmel used in this paper, cf. [3, 7, 12]. In Section 3 the abstract domains underlying the analysis are examined. Following that the control flow analysis and specification are discussed in Section 4, while Section 5 is devoted to formal theoretical results concerning the analysis. Section 6 concludes the document by discussing a number of directions for future work, including enhancements, formal results, and applications of the analysis.

2 The Language

The language of interest in this paper is an abstraction or a "rational reconstruction" of JCVML as described and defined in [7]. This language is called Carmel and in [12] a small step semantics is given for it. Carmel is an abstraction, or a rational reconstruction, of JCVML in the sense that a number of instructions have been generalised and/or merged, thus reducing the overall number of instructions to 30 while retaining the expressiveness of the original JCVML.

Although the work in this paper is in part motivated by and based on the work of Freund and Mitchell, the analysis we present is a naïve *inter*procedural analysis whereas the typesystem of Freund and Mitchell is *intra*procedural, cf. Appendix A of [7]. The analysis is naïve in the sense that method invocations and returns are handled much like unconditional jumps. A less naïve treatment can be obtained by applying standard techniques, cf. [9].

In what follows, a number of technical details have been left out, such as specific representation of programs and packages, pending finalisation of the semantics for these.

3 Abstract Domains

The abstract domains are based on a simplified version of the concrete domains used in the semantics, cf. [12]. The simplified domains ignore semantic information that is not pertinent to the analysis. This minimises unnecessary notation and increases legibility of both analysis and theoretical results.

For the simple analysis, objects are abstracted into their class, thus object references are modeled as classes, similar to the *class object graphs* of [14]:

$$\mathsf{ObjRef} = \mathsf{Class}$$

In order to enhance readability we write (Ref σ), rather than merely σ, for object references. Similarly arrays are abstracted into their elementtype:

$$\mathsf{ArRef} = \mathsf{Type}$$

We write (Ref (`array` τ)) rather than τ for array references. Note that (Ref σ) is used exclusively for *object* references and (Ref (`array` τ)) exclusively for *array* references.

References are either object references or array references:

$$\mathsf{Ref} = \mathsf{ObjRef} + \mathsf{ArRef}$$

An obvious improvement to the precision of the analysis would be to model object and array references using the equivalent of the *textual object graphs* as described in [14], however, in order to prove the correctness of the analysis, it would then require additional information in the semantics about where objects and arrays are created.

In Java Card applications (ie. applets) objects and arrays are usually only created once, during the installation, and then reused throughout the lifetime of the applet, cf. [1, 6]. This would seem to make (the equivalent of) textual object graphs ideally suited for the analysis of applets.

Following the semantics and given the fact that subroutines (and thus return addresses) are not needed, values are taken to be either numerical values or reference values and abstract values to be sets of such values:

$$\mathsf{Val} = \mathsf{Num} + \mathsf{Ref} \qquad \widehat{\mathsf{Val}} = \mathcal{P}(\mathsf{Val})$$

Objects themselves are modeled as mappings from the field ID's of the object, to the set of abstract values possibly contained in that field

$$\widehat{\mathsf{Obj}} = \mathsf{FieldID} \to \widehat{\mathsf{Val}}$$

Arrays are modeled in the simplest possible way, namely as an abstract value. This means that the structure (and length) of the array is abstracted away:

$$\widehat{\mathsf{Array}} = \widehat{\mathsf{Val}}$$

While other, more precise, abstract representations of arrays are possible, they would also require a more precise analysis of integer values used as indices for arrays, possibly even a full dataflow analysis. For this reason we have chosen the above representation as basis for the simple control flow analysis described in this paper.

Adresses consist of a method and a program counter, making adresses unique in a program. In order to correctly handle return values from method invocations a special "placeholder" address is defined for every method. This placeholder address is encoded using a special END-token instead of the regular program counter.

$$\mathsf{Addr} = \mathsf{Method} \times (\mathbb{N} \uplus \{\mathsf{END}\})$$

The first instruction in a method is assumed to be at program counter 1 and we write (m, END_m) for the placeholder address belonging to the method m.

We let $|m|$ denote the arity of method m, meaning the number of arguments the method expects on the operand stack.

In order to model the memory organisation of the Java (Card) Virtual Machine, we must model the local heap (local variables for each method), the operand stack (also one for each method) and the global heap.

The local heap is modeled as a (curried) map from adresses to (local) variables to abstract values. Thus in our model there is a local heap associated with *every instruction* in a method. This is in keeping with Freund and Mitchells approach, cf. [3]. Less precise, and less costly, analyses are possible, for instance by merging the analysis information for all instructions in a given method. Other merging strategies, eg. merging at basic blocks, could be considered for balancing cost and precision.

$$\widehat{\mathsf{LocHeap}} = \mathsf{Addr} \to \mathsf{Var} \to \widehat{\mathsf{Val}}$$

For $\hat{L} \in \widehat{\mathsf{LocHeap}}$ we shall write $\hat{L}(m_1, pc_1) \sqsubseteq \hat{L}(m_2, pc_2)$ to mean

$$\forall x \in \mathrm{dom}(\hat{L}(m_1, pc_1)) : \hat{L}(m_1, pc_1)(x) \sqsubseteq \hat{L}(m_2, pc_2)(x)$$

and $\hat{L}(m_1, pc_1) \sqsubseteq_{\{x\}} \hat{L}(m_2, pc_2)$ to mean

$$\forall y \in \mathrm{dom}(\hat{L}(m_1, pc_1)) \setminus \{x\} : \hat{L}(m_1, pc_1)(y) \sqsubseteq \hat{L}(m_2, pc_2)(y)$$

Note that local variables are denoted by natural numbers and zero, ie. $\mathsf{Var} = \mathbb{N}_0$.

We now turn to the operand stack. Since the model has to be able to cope with potentially infinite operand stacks, we use the following domain as the basis for the stack model:

$$\widehat{\mathsf{Val}}^\infty = \widehat{\mathsf{Val}}^\omega \cup \widehat{\mathsf{Val}}^*$$

However, in anticipation of later developments and applications of the analysis, rather than using the above domain directly, we use it to induce a more convenient domain via a Galois connection:

$$\widehat{\mathsf{Val}}^\infty \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathsf{Val}}^*)^\top$$

where the abstraction function, α, simply acts as the identity on finite stacks and maps infinite stacks to top.

With the basic domain for abstract stacks in place, we now associate an abstract operand stack with every instruction in a method in order to track operations on the stack in that method:

$$\widehat{\mathsf{Stack}} = \mathsf{Addr} \to (\widehat{\mathsf{Val}}^*)^\top$$

Elements of $(\widehat{\mathsf{Val}^*})^\top$ are written much in the same style as SML lists, thus $(A_1 :: A_2 :: \cdots :: X) \in (\widehat{\mathsf{Val}^*})^\top$ represents a stack with $A_1 \in \widehat{\mathsf{Val}}$ as its top element and $X \in \widehat{\mathsf{Val}}^\omega$ as the "bottom" of the stack. The empty stack is denoted by ϵ. We introduce the following ordering on abstract stacks, $A_1 :: \cdots :: A_n$ and $B_1 :: \cdots :: B_m$, from $(\widehat{\mathsf{Val}^*})^\top$:

$$\begin{aligned}(A_1 :: \cdots :: A_n) \sqsubseteq (B_1 :: \cdots :: B_m) \iff \\ m \geq n \wedge \forall i \in \{1, \ldots, n\} : A_i \subseteq B_i\end{aligned}$$

In the interest of succinctness we shall abuse the above notation slightly by writing $(A_0 :: \cdots :: A_n) \sqsubseteq \hat{L}(m_0, pc_0)[0..n]$ as a shorthand for

$$\forall i \in \{0, \ldots, n\} : A_i \subseteq \hat{L}(m_0, pc_0)(i)$$

The abstract global heap comprises two components: an object component, keeping track of instance fields of individual objects, and a static component, that tracks the values of static fields for each class. The static component is taken to be a map from field ID's to abstract values:

$$\widehat{\mathsf{StaHeap}} = \mathsf{FieldID} \to \widehat{\mathsf{Val}}$$

and the object component is taken to be a map from references to abstract objects and arrays:

$$\widehat{\mathsf{Heap}} = (\mathsf{ObjRef} \to \widehat{\mathsf{Obj}}) + (\mathsf{ArRef} \to \widehat{\mathsf{Array}})$$

By simple changes of the definitions of adresses and object references, a variety of analyses are possible ranging from the rather cheap and imprecise to the prohibitively expensive but very precise analyses.

4 Flow Logic Specification

In this section we describe and discuss the Flow Logic specification of a simple Control Flow Analysis of Carmel. First we discuss the general form of the specification. Following that a few clauses are discussed at length and finally the full specification is given. The full specification is split into several parts, corresponding to the language hierarchy described in [5] with the exception that we do not distinguish between single-word and multi-word instructions.

The judgements and clauses discussed in the following all concentrate on specifying an acceptable analysis for single instructions. While not formalised yet, the intention is that a program (or a package) consists of a number of classes, each containing a number of methods. In order for an analysis to be acceptable with respect to a package, it should be acceptable with respect to every instruction (as detailed in the following sections) in every method in every class in the package.

4.1 The Judgements

The Flow Logic framework can be seen as a "specification approach" to static analysis, rather than an "implementation approach". In the framework, rather

than detailing how a particular static analysis is to be carried out, it is specified what it means for an analysis result (or rather a *proposed* analysis result) to be acceptable (correct) with respect to a program. Flow Logic specifications are usually classified as either *verbose* or *succinct* according to the style of specification: succinct resembling the style of type-systems in only reporting "top-level" information and verbose more like traditional data flow and constraint based analyses in recording all internal flows. The specification in this paper is a verbose specification. We shall not go into further detail with the framework here, merely refer to [8] and [11] for further information.

The judgements of the Flow Logic specification for the analysis of Carmel will be on the form

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models \texttt{addr} : \texttt{instr}$$

where $\hat{S} \in \widehat{\mathsf{Stack}}$, $\hat{L} \in \widehat{\mathsf{LocHeap}}$, $\hat{H} \in \widehat{\mathsf{Heap}}$, $\hat{K} \in \widehat{\mathsf{StaHeap}}$, $\texttt{addr} \in \mathsf{Addr}$ and `instr` is the instruction at `addr`. Intuitively the above states that $(\hat{K}, \hat{H}, \hat{L}, \hat{S})$ is an *acceptable* analysis for the instruction `instr` at address `addr`. A detailed discussion of the clauses and judgements for Carmel are given in the following sections.

Note that while some of the Carmel instructions explicitly give the type(s) of their argument(s), this information is ignored in the current analysis. We believe that in a later stage, the analysis can, and should, be made more precise by also taking type information into account.

4.2 A Few Interesting Clauses

In this section we discuss a number of clauses in great detail, so as to explain and discuss the used notation and conventions. This should also give an insight into how the analysis works. In Section 4.3 the full specification is given.

4.2.1 The `push` Instruction

We wish to specify the analysis for the `push` instruction, ie. we wish to specify the following:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{push}\ t\ v$$

The `push` instruction simply pushes its argument, v, onto the top of the current stack, $\hat{S}(m_0, pc_0)$, giving rise to a new stack: $\{v\} :: \hat{S}(m_0, pc_0)$. The new stack is then made available to the next instruction (at $(m_0, pc_0 + 1)$), thus:

$$\{v\} :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1)$$

Furthermore, the `push` instruction does not alter any local variables, therefore the local heap is made available to the next instruction with no modification:

$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$

Combining the above we have the following specification for the `push` instruction:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{push}\ t\ v \\ \quad \text{iff} \quad \{v\} :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ \qquad\quad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)\end{array}$$

4.2.2 The `store` Instruction

The next instruction of interest is the `store` instruction:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{store}\ t\ x$$

The `store` instruction saves the top element of the stack in the variable given as argument to the instruction. In order for this to work, there must be at least one element on the stack. We introduce the notation

$$A :: X \triangleleft \hat{S}(m_0, pc_0) :$$

as a succint way of expressing that the abstract stack should be on the form $A :: X$, ie. it should have at least one element. Furthermore, it acts as a binder for the variables A and X so that they may subsequently be referred to.

The `store` instruction transfers the bottom of the stack to the stack of the next instruction: $X \sqsubseteq \hat{S}(m_0, pc_0 + 1)$. The top element is stored in the local heap (for the next instruction) at the variable given as argument to the `store` instruction: $A \sqsubseteq \hat{L}(m_0, pc_0 + 1)(x)$. Finally the variables in the local heap that were not modified by the instruction (all but x) are transferred to the next instruction: $\hat{L}(m_0, pc_0) \sqsubseteq_{\{x\}} \hat{L}(m_0, pc_0 + 1)$.

Putting all of the above together we arrive at the clause below for `store` instructions

$$\begin{array}{rl} (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{store}\ t\ x & \\ \quad \text{iff} & A :: X \triangleleft \hat{S}(m_0, pc_0) : \\ & X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ & A \sqsubseteq \hat{L}(m_0, pc_0 + 1)(x) \\ & \hat{L}(m_0, pc_0) \sqsubseteq_{\{x\}} \hat{L}(m_0, pc_0 + 1) \end{array}$$

4.2.3 The `putfield` Instruction

Next is the specification for the `putfield` instruction:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield}\ f$$

The `putfield` instruction transfers the value of the top element of the stack to the field named as argument to the instruction in the object referenced in the second element of the stack. Thus the stack must contain at least two elements:

$$A :: B :: X \triangleleft \hat{S}(m_0, pc_0) :$$

The specific object to be accessed is resolved at runtime, and a reference to that object is stored in the second (from the top) element of the stack. The value of the top element is then stored in the field of the object so referenced:

$$\forall(\text{Ref}\ \sigma') \in B : A \sqsubseteq \hat{H}(\text{Ref}\ \sigma')(f.\text{id})$$

Here we use the abstract global heap to hold information about the fields of abstract objects. As noted in Section 3 objects are abstracted into their class. Thus field information for all objects of the same class is merged and stored in the abstract global heap.

The bottom of the stack is then transferred to the next instruction:

$$X \sqsubseteq \hat{S}(m_0, pc_0 + 1)$$

and since no local variables were modified, the abstract local heap is transferred unchanged to the next instruction

$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$

We then arrive at the following clause for `putfield` instructions:

$$\begin{array}{ll}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield } f \\ \quad \text{iff} & A :: B :: X \lhd \hat{S}(m_0, pc_0) : \\ & \forall(\text{Ref } \sigma') \in B : A \sqsubseteq \hat{H}(\text{Ref } \sigma')(f.\text{id}) \\ & X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ & \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)\end{array}$$

4.2.4 The `invokevirtual` Instruction

Finally we discuss the specification for the `invokevirtual` instruction:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual } m$$

In order to call an instance method, the `invokevirtual` instruction is used. Arguments to the method is found at the top of the stack, and as was the case for the `putfield` instruction, a reference to the specific object containing the invoked method is found on the stack, immediately following the arguments to the method:

$$A_1 :: \cdots :: A_{|m|} :: B :: X \lhd \hat{S}(m_0, pc_0) :$$

Next a method lookup is needed in order to find the actual method that is executed:

$$m_v = \mathit{methodLookup}(m.\text{id}, \sigma')$$

The arguments are transferred to the called method as *local variables* of the called method. Furthermore a reference to object containing the called method is passed as the first local variable (in effect a `this` pointer):

$$\{(\text{Ref } \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|]$$

When a method invocation returns, there are two possibilities: either it does not return a value, ie. it has return type `void`, or it does return a value. In the first case, m.returnType = `void`, we simply copy the rest of the stack on to the next instruction:

$$\begin{array}{c}m.\text{returnType} = \texttt{void} \Rightarrow \\ X \sqsubseteq \hat{S}(m_0, pc_0 + 1)\end{array}$$

In the latter case, m.returnType $\neq$ `void`, the return value is the top element of the stack of the invoked method. In order to handle multiple returns from the invoked method correctly a special address is used, indicated by the END-token discussed in Section 3; it is the responsibility of the clause for the `return` instruction to ensure, that all the possible stacks at all possible `return` instructions are transferred to the stack at the special address.

In order for the invoking method to access the return value, it must be transferred from the top of the stack of the *invoked* method to the top of the stack of the *invoking* method (less the arguments and the object reference):

$$\begin{array}{l}m.\text{returnType} \neq \texttt{void} \Rightarrow \\ \quad T :: Y \lhd \hat{S}(m_v, \mathsf{END}_{m_v}) : \; T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1)\end{array}$$

Finally, none of the local variables (of the invoking method) have been altered and are therefore passed on to the next instruction:

$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$

Joining the above equations we obtain the following clause for `invokevirtual` instructions:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual}\ m \\ \quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\ \qquad \forall (\text{Ref}\ \sigma') \in B : \\ \qquad\quad m_v = \mathit{methodLookup}(m.\text{id}, \sigma') \\ \qquad\quad \{(\text{Ref}\ \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|] \\ \qquad\quad m.\text{returnType} \neq \texttt{void} \Rightarrow \\ \qquad\qquad T :: Y \triangleleft \hat{S}(m_v, \mathsf{END}_{m_v}) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ \qquad\quad m.\text{returnType} = \texttt{void} \Rightarrow \\ \qquad\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ \qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)\end{array}$$

4.3 Full Specification

In this section the Flow Logic specification for all the supported instructions is given. It is divided into categories matching those of [5] with the exceptions noted in beginning of Section 4.

4.3.1 Core Language

The Core Language consists of instructions for stack management, arithmetic operations, control transfer and local variable access.

The basic stack management instructions considered are: `push`, `pop`, `dup`, and `swap`. Note that the `push` instruction gives an explicit type for its argument: `t`. This type-information is currently ignored by the analysis, as explained in Section 4.1.

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{push}\ \ t\ \ v \\ \quad \text{iff} \quad \{v\} :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ \qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)\end{array}$$

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{pop}\ \ n \\ \quad \text{iff} \quad A_1 :: \cdots :: A_n :: X \triangleleft \hat{S}(m_0, pc_0) : \\ \qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ \qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)\end{array}$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{dup}\ \ m\ \ n \\
\quad \text{iff} \quad S_1 :: S_2 :: X \lhd \hat{S}(m_0, pc_0) : \\
\qquad A_1 :: \cdots :: A_m \lhd S_1 : \\
\qquad A_{m+1} :: \cdots :: A_n \lhd S_2 : \\
\qquad S_1 :: S_2 :: S_1 :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{swap}\ \ m\ \ n \\
\quad \text{iff} \quad S_1 :: S_2 :: X \lhd \hat{S}(m_0, pc_0) : \\
\qquad A_1 :: \cdots :: A_m \lhd S_1 : \\
\qquad A_{m+1} :: \cdots :: A_n \lhd S_2 : \\
\qquad S_2 :: S_1 :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

The δ function below is the abstract equivalent of the arithmetic operators. In particular, δ_{unop} is the abstract representation of the unary operator *unop* and similarly for binary operators. The exact definition of δ depends on how precise the data flow needs to tracked. It is assumed that both functions are total. Error conditions are signaled by throwing an exception.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{numop}\ \ t\ \ unop\ \ t'_{opt} \\
\quad \text{iff} \quad A :: X \lhd \hat{S}(m_0, pc_0) : \\
\qquad \delta_{unop}(A) :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{numop}\ \ t\ \ binop\ \ t'_{opt} \\
\quad \text{iff} \quad A_1 :: A_2 :: X \lhd \hat{S}(m_0, pc_0) : \\
\qquad \delta_{binop}(A_1, A_2) :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

In the above, the explicitly arugment types are ignored as explained earlier.

Conditional and unconditional branching is taken care of by the next few instructions. Note that the analysis of the conditional branching instructions simply assume that control can flow to all branches. This conservative, and rather imprecise, etsimate can be mitigated by taking data flow into account. Below there are two variants of the `if` instruction: one variant compares the two top elements on the stack, and the other variant compares the top element

on the stack to either `0` or `null`.

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{goto}\ L$$
$$\text{iff} \quad \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, L)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, L)$$

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{if}\ t\ \mathit{cmpop}\ \texttt{goto}\ L$$
$$\text{iff} \quad A_1 :: A_2 :: X \triangleleft \hat{S}(m_0, pc_0) :$$
$$X \sqsubseteq \hat{S}(m_0, pc_0 + 1)$$
$$X \sqsubseteq \hat{S}(m_0, L)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, L)$$

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{if}\ t\ \mathit{cmpop}\ \mathit{nullCmp}\ \texttt{goto}\ L$$
$$\text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) :$$
$$X \sqsubseteq \hat{S}(m_0, pc_0 + 1)$$
$$X \sqsubseteq \hat{S}(m_0, L)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, L)$$

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{lookupswitch}\ t\ (k_i\texttt{=>}L_i)_1^n,\ \texttt{default=>}L_0$$
$$\text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) :$$
$$\forall i \in \{0, 1, \ldots, n\} :$$
$$X \sqsubseteq \hat{S}(m_0, L_i)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, L_i)$$

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{tableswitch}\ t\ l\texttt{=>}(L_i)_0^n,\ \texttt{default=>}L_{n+1}$$
$$\text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) :$$
$$\forall i \in \{0, \ldots, n, n+1\} :$$
$$X \sqsubseteq \hat{S}(m_0, L_i)$$
$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, L_i)$$

The next batch of instructions handle access to local variables. This includes instructions for loading and storing values in local variables and also for incre-

menting the numerical value of a local variable.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{load}\ t\ x \\
\quad \text{iff} \quad \hat{L}(m_0, pc_0)(x) :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{store}\ t\ x \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad A \sqsubseteq \hat{L}(m_0, pc_0 + 1)(x) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq_{\{x\}} \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{inc}\ t\ x\ c \\
\quad \text{iff} \quad \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \delta_{inc}(\hat{L}(m_0, pc_0)(x), \{c\}) \sqsubseteq \hat{L}(m_0, pc_0 + 1)(x) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq_{\{x\}} \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

4.3.2 Object Language

Instructions for object creation and access to instance fields belong to the "Object Language".

First the instructions for performing explicit runtime typechecks. Note that because the result of an `instanceof` instruction is either 0 or 1, the analysis models this by pushing both 0 and 1 onto the stack. Furthermore these instructions act like conditional branching instrcutions and are analysed as such, ie. the arguments for the instructions are ignored.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{checkcast}\ t \\
\quad \text{iff} \quad \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{instanceof}\ t \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \{0, 1\} :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

Next the instruction for creating new instances of classes, ie. objects:

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{new}\ \sigma \\
\quad \text{iff} \quad \{(\text{Ref}\ \sigma)\} :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \text{default}(\sigma) \sqsubseteq \hat{H}(\text{Ref}\ \sigma) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

where default(σ) is defined such that

$$
\begin{array}{l}
\forall f \in \text{instanceFields}(\sigma) : \\
\quad \left\{ \begin{array}{ll}
\beta^H(\text{def}(f.\text{type})) \sqsubseteq \text{default}(\sigma)(f.\text{id}) & \text{if } f.\text{initValue} = \bot \\
\beta^H(f.\text{initValue}) \sqsubseteq \text{default}(\sigma)(f.\text{id}) & \text{if } f.\text{initValue} \neq \bot
\end{array} \right.
\end{array}
$$

The above makes sure that the fields in a new object are initialised with their explicit, as indicated in the program, initial values or their default values if no explicit initial value is given.

The next clauses deal with access to instance fields. In [7] the instructions for accessing instance fields have an optional `this` modifier. The effect of such a modifier is to access the fields of the particular object that contains the current method, rather than an object referenced on the top of the stack. We deal with both cases in separate clauses below.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{getfield } f \\
\quad \text{iff} \quad B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } \sigma') \in B : (\hat{H}(\text{Ref } \sigma')(f.\text{id})) :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{getfield this } f \\
\quad \text{iff} \quad \forall(\text{Ref } \sigma') \in \hat{L}(m_0, pc_0)(0) : \\
\qquad (\hat{H}(\text{Ref } \sigma')(f.\text{id})) :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield } f \\
\quad \text{iff} \quad A :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } \sigma') \in B : A \sqsubseteq \hat{H}(\text{Ref } \sigma')(f.\text{id}) \\
\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield this } f \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } \sigma') \in \hat{L}(m_0, pc_0)(0) : A \sqsubseteq \hat{H}(\text{Ref } \sigma')(f.\text{id}) \\
\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

Access to static fields (class fields) is handled in much the same way, except that only the field identifier is needed rather than the full object reference:

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{getstatic } f \\
\quad \text{iff} \quad \hat{K}(f.\text{id}) :: \hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putstatic } f \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad A \sqsubseteq \hat{K}(f.\text{id}) \\
\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

4.3.3 Method Support

The "Method Support" fragment of the language is made up of instructions for invoking, and returning from, methods.

We first consider the specification for the `invokedefinite` instruction. For convenience of notation we split the `invokedefinite` instruction into two instructions, `invokestatic` and `invokespecial`, representing the fact that the

semantics of `invokedefinite` depends on which (type of) method it is used to invoke (static or special).

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokestatic}\ m \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m, 1)[0..|m| - 1] \\
\qquad m.\text{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\quad T :: Y \triangleleft \hat{S}(m, \mathsf{END}_m) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad m.\text{returnType} = \texttt{void} \Rightarrow \\
\qquad\quad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokespecial}\ m \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad B :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m, 1)[0..|m|] \\
\qquad m.\text{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\quad T :: Y \triangleleft \hat{S}(m, \mathsf{END}_m) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad m.\text{returnType} = \texttt{void} \Rightarrow \\
\qquad\quad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

Next the clause for handling the invocation of virtual methods. Note that we explicitly distinguish between methods that return a value, ie. those methods with $m.\text{returnType} \neq \texttt{void}$, and those that do not, ie. where $m.\text{returnType} = \texttt{void}$.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual}\ m \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } \sigma') \in B : \\
\qquad\quad m_v = \mathit{methodLookup}(m.\text{id}, \sigma') \\
\qquad\quad \{(\text{Ref } \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|] \\
\qquad\quad m.\text{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad T :: Y \triangleleft \hat{S}(m_v, \mathsf{END}_{m_v}) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad\quad m.\text{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

The clause for `invokeinterface` is the same as the clause for `invokevirtual`. This is due to the fact that regarding the flow of control there is no real difference between interface and virtual methods.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokeinterface}\ m \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } \sigma') \in B : \\
\qquad\quad m_v = \mathit{methodLookup}(m.\text{id}, \sigma') \\
\qquad\quad \{(\text{Ref } \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|] \\
\qquad\quad m.\text{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad T :: Y \triangleleft \hat{S}(m_v, \mathsf{END}_{m_v}) : T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad\quad m.\text{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

And finally the clauses for returning from a method invocation. Again there are two cases: one that does not return a value and one that does. The case where no value is returned is trivial.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{return} \\
\quad \text{iff} \quad \mathit{true}
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{return } t \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad A :: X \sqsubseteq \hat{S}(m_0, \mathsf{END}_{m_0})
\end{array}
$$

This ends the discussion of the "Method Support".

4.3.4 Array Support

Array support in Carmel consists of instructions for creating arrays, calculating the length of an array and for loading and storing values in arrays.

The abstract representation of arrays simply ignores the structure of the array and treats the array as a simple variable. While rudimentary, this view of arrays is sufficient for analysing the control flow. By adding a dataflow component to the control flow analysis, more sophisticated analyses of arrays would be possible.

First the instruction for creating a new array. Note that the value on top of the stack, the length of the array to be created, is ignored:

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{new (array } \tau\texttt{)} \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \{(\text{Ref } (\texttt{array } \tau))\} :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

Next we have the clauses for handling array length and manipulation of data in an array. As before, type information is ignored.

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{arraylength} \\
\quad \text{iff} \quad B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \{\mathsf{INT}\} :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{arrayload } t \\
\quad \text{iff} \quad A :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } (\texttt{array } \tau)) \in B : \\
\qquad\quad \hat{H}(\text{Ref } (\texttt{array } \tau)) :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{arraystore } t \\
\quad \text{iff} \quad A_1 :: A_2 :: B :: X \triangleleft \hat{S}(m_0, pc_0) : \\
\qquad \forall(\text{Ref } (\texttt{array } \tau)) \in B : \\
\qquad\quad A_1 \sqsubseteq \hat{H}(\text{Ref } (\texttt{array } \tau)) \\
\qquad\quad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

Note that for `arrayload` and `arraystore` the index into the array where a value is to be loaded from (A) or stored to (A_2) respectively, is ignored. A more precise analysis of arrays would be possible if a dataflow component was added to the control flow analysis.

5 Theoretical Results

5.1 Semantic Soundness

In this section we establish the semantic soundness of the analysis. Soundness is proved with respect to a slightly simplified version of the semantics defined in [12]. The simplification is achieved by ignoring the information in the semantics that is not used by the analysis, eg. owner context and type information. The notation has been adapted accordingly.

The soundness is proved using *representation functions*, cf. [9] for a discussion of representation functions. The next section will define representation functions for the domains used in the (modified) semantics and the analysis. The section following that will state the soundness theorem (in the form of a subject reduction result) and detail a few cases in the proof.

5.1.1 Representation Functions

Values. The representation function for integer values basically injects the number into a set:

$$\beta_{\mathrm{Num}}(n) = \{n\}$$

The notion of a location only makes sense relative to a given heap, thus the representation function for locations is parameterised on a heap, H:

$$\beta^H_{\mathrm{Ref}}(loc) = \begin{cases} H(loc).\mathrm{class} & \text{if } H(loc) \in \mathsf{Object} \\ H(loc).\mathrm{elementType} & \text{if } H(loc) \in \mathsf{Array} \end{cases}$$

Values (in the modified semantics) can be either numbers or references, the representation function for values acts accordingly:

$$\beta^H_{\mathrm{Val}}(v) = \begin{cases} \beta_{\mathrm{Num}}(v) & \text{if } v \in \mathsf{Num} \\ \beta^H_{\mathrm{Ref}}(v) & \text{if } v \in \mathsf{Ref} \end{cases}$$

Note that because the representation function for values depends on the representation function for references, the representation function for values also takes a global heap as parameter. This will be the case for any representation function that depends on the representation function for values (or references).

We introduce a simple partial order on abstract values: Let $\hat{v}_1, \hat{v}_2 \in \widehat{\mathsf{Val}}$ and define

$$\hat{v}_1 \sqsubseteq \hat{v}_2 \quad \text{iff} \quad \hat{v}_1 \subseteq \hat{v}_2$$

Stacks. Stacks are sequences of values and the abstract representation is simply a sequence of the corresponding abstract values. Let $S \in \mathsf{Stack}$ such that $S = v_1 :: \cdots :: v_n$.

$$\beta^H_{\mathrm{Stack}}(S) = \beta^H_{\mathrm{Val}}(v_1) :: \cdots :: \beta^H_{\mathrm{Val}}(v_n)$$

A partial order is also introduced on the abstract stacks. Let $\hat{S}_1, \hat{S}_2 \in \widehat{\mathsf{Stack}}$ and $addr \in \mathsf{Addr}$ such that $\hat{S}_1(addr) = \hat{v}_1 :: \cdots :: \hat{v}_m$ and $\hat{S}_2(addr) = \hat{v}'_1 :: \cdots :: \hat{v}'_n$ then

$$\hat{S}_1(addr) \sqsubseteq \hat{S}_2(addr) \quad \text{iff} \quad m \leq n \wedge \forall i \in \{1, \ldots, m\} : \hat{v}_i \sqsubseteq \hat{v}'_i$$

Local Variables. The abstract representation of local variables (the local heap) maps variables to the corresponding abstract value: Let $V \in \mathsf{LocalVar}$, then define

$$\beta^H_{\text{LocalVar}}(V) = \beta^H_{\text{Val}} \circ V$$

thus in particular $\forall x \in \mathsf{Var} : \beta^H_{\text{LocalVar}}(V)(x) = \beta^H_{\text{Val}}(V[x])$.

We introduce a pointwise ordering on the abstract local heap. Now let $\hat{L}_1, \hat{L}_2 \in \widehat{\mathsf{LocHeap}}$ and $addr \in \mathsf{Addr}$ then

$$\begin{aligned} \hat{L}_1(addr) \sqsubseteq \hat{L}_2(addr) \quad \text{iff} \quad & \text{dom}(\hat{L}_1(addr)) \subseteq \text{dom}(\hat{L}_2(addr)) \wedge \\ & \forall x \in \text{dom}(\hat{L}_1(addr)) : \hat{L}_1(addr) \sqsubseteq \hat{L}_2(addr) \end{aligned}$$

Objects. Abstract objects are mappings from fields (fieldnames) to abstract values, thus the representation function for objects abstracts away all other information in the object: Let $o \in \mathsf{Object}$ and define

$$\beta^H_{\text{Object}}(o) = \beta^H_{\text{Val}} \circ (o.\text{fieldValue})$$

and therefore in particular $\beta^H_{\text{Object}}(o).f = \beta^H_{\text{Val}}(o.\text{fieldValue}(f.id))$. Again we use a pointwise ordering for abstract objects. Let $\hat{o}_1, \hat{o}_2 \in \widehat{\mathsf{Obj}}$ then

$$\hat{o}_1 \sqsubseteq \hat{o}_2 \quad \text{iff} \quad \text{dom}(\hat{o}_1) \subseteq \text{dom}(\hat{o}_2) \wedge \forall f \in \text{dom}(\hat{o}_1) : \hat{o}_1(f) \sqsubseteq \hat{o}_2(f)$$

Arrays. The abstract representation of arrays is simply as an abstract value, with no additional structure. The representation function therefore abstracts all the values in an array into one abstract value. Let $a \in \mathsf{Array}$ and define

$$\beta^H_{\text{Array}}(a) = \bigsqcup_{0 \leq i < a.\text{length}} \beta^H_{\text{Val}}(a.\text{values}(i))$$

The ordering on abstract arrays is the same as for abstract values. Let $a_1, a_2 \in \widehat{\mathsf{Array}}$ then

$$a_1 \sqsubseteq a_2 \quad \text{iff} \quad a_1 \subseteq a_2$$

Heaps. The global heap maps locations to objects or arrays. Because locations are represented in the analysis simply by the class of object or the type of array they point to in the heap, an abstract location may represent several different concrete locations. For this reason, the abstract representation of a heap is the least upper bound of all the concrete objects and arrays having the same abstract location, ie. all those objects and arrays that belong to the same class.

Let $H \in \mathsf{Heap}$ and define

$$\beta_{\text{Heap}}(H)(\text{Ref } \sigma) = \bigsqcup_{\substack{loc \in \text{dom}(H) \\ \beta^H_{\text{Ref}}(loc) = (\text{Ref } \sigma)}} \beta^H_{\text{Object}}(H(loc))$$

and similarly

$$\beta_{\text{Heap}}(H)(\text{Ref } (\texttt{array } \tau)) = \bigsqcup_{\substack{loc \in \text{dom}(H) \\ \beta^H_{\text{Ref}}(loc) = (\text{Ref } (\texttt{array } \tau))}} \beta^H_{\text{Array}}(H(loc))$$

Now for $\hat{H}_1, \hat{H}_2 \in \widehat{\mathsf{Heap}}$ we define the following partial order on abstract heaps:

$$\begin{aligned} \hat{H}_1 \sqsubseteq \hat{H}_2 \quad \text{iff} \quad & \text{dom}(\hat{H}_1) \subseteq \text{dom}(\hat{H}_2) \wedge \\ & \forall (\text{Ref } \sigma) \in \text{dom}(\hat{H}_1) : \hat{H}_1(\text{Ref } \sigma) \sqsubseteq \hat{H}_2(\text{Ref } \sigma) \end{aligned}$$

Static Memory. The static memory, or the static heap, holds information on the static fields of classes, thus it maps (static) field id's to values. This gives rise to the following representation function for static memory:

$$\beta^H_{\text{StatMem}}(K) = \beta^H_{\text{Val}} \circ K$$

The obvious partial order is induced on abstract static memories. Let $\hat{K}_1, \hat{K}_2 \in \widehat{\mathsf{StatMem}}$ then

$$\begin{aligned} \hat{K}_1 \sqsubseteq \hat{K}_2 \quad \text{iff} \quad & \text{dom}(\hat{K}_1) \subseteq \text{dom}(\hat{K}_2) \wedge \\ & \forall \mathit{fid} \in \text{dom}(\hat{K}_1) : \hat{K}_1(\mathit{fid}) \sqsubseteq \hat{K}_2(\mathit{fid}) \end{aligned}$$

Call Stack. Call stacks represent the control flow for Carmel programs. Each stack frame contains information on the method that "owns" it, the current program counter in that method, the local heap for that method and finally the operand stack for the owning method: Let $SF = \langle m_1, pc_1, V_1, S_1 \rangle :: \cdots :: \langle m_n, pc_n, V_n, S_n \rangle$ be such a call stack, we then define the abstract representation by applying the relevant representation functions pointwise:

$$\beta^H_{\text{CallStack}}(SF) = (\beta^H_{\text{LocalVar}}(V_1), \beta^H_{\text{Stack}}(S_1)) :: \cdots :: (\beta^H_{\text{LocalVar}}(V_n), \beta^H_{\text{Stack}}(S_n))$$

We also define what it means for such an abstract call stack to be represented in the analysis, formalised by the $\hat{\mathcal{R}}_{\text{CallStack}}$ relation:

$$\begin{aligned} & \beta^H_{\text{CallStack}}(SF)\, \hat{\mathcal{R}}_{\text{CallStack}}\, (\hat{S}, \hat{L}) \quad \text{iff} \\ & \quad \forall i \in \{1, \ldots, n\} : \beta^H_{\text{LocalVar}}(V_i) \sqsubseteq \hat{L}(m_i, pc_i) \wedge \beta^H_{\text{Stack}}(S_i) \sqsubseteq \hat{S}(m_i, pc_i) \end{aligned}$$

Frames. A frame is the basic configuration of the semantics. The abstract representation of such a frame consists of pointwise application of the appropriate representation function on the components of the frame:

$$\beta_{\text{Frame}}(\langle K, H, SF \rangle) = (\beta^H_{\text{StatMem}}(K), \beta_{\text{Heap}}(H), \beta^H_{\text{CallStack}}(SF))$$

and again we formalise by the relation $\hat{\mathcal{R}}_{\text{Frame}}$ what it means for an abstract frame to be represented:

$$\begin{aligned} & \beta_{\text{Frame}}(\langle K, H, SF \rangle)\, \hat{\mathcal{R}}_{\text{Frame}}\, (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \quad \text{iff} \\ & \quad \beta^H_{\text{StatMem}}(K) \sqsubseteq \hat{K} \wedge \beta_{\text{Heap}}(H) \sqsubseteq \hat{H} \wedge \beta^H_{\text{CallStack}}(SF)\, \hat{\mathcal{R}}_{\text{CallStack}}\, (\hat{L}, \hat{S}) \end{aligned}$$

In the following we shall dispense with the subscripts on representation functions.

5.1.2 Subject Reduction

In this section we formally state and prove the basic soundness result for the analysis. Following the Flow Logic framework this is done by proving a subject reduction result. In order to prove subject reduction we first need to prove a technical lemma that characterises well-formed reduction sequences in the semantics. We start by defining what it means for a configuration in the semantics to be well-formed:

Definition 1 (Well-Formedness) *A frame,* $\langle K, H, \langle m_1, pc_1, V_1, S_1\rangle :: \cdots :: \langle m_n, pc_n, V_n, S_n\rangle\rangle$*,is said to be* well-formed *if and only if*

$$\forall i \in \{2,\ldots,n\}: \left(\begin{array}{l} m_i.\text{instructionAt}(pc_i) = \texttt{invokevirtual}\ m_i'' \wedge \\ S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: loc_i :: S_i' \wedge \\ H(loc_i).\text{class} = \sigma_i \wedge \\ m_{i-1} = \text{methodLookup}(m_i''.\text{id}, \sigma_i) \end{array}\right) \vee$$
$$\left(\begin{array}{l} m_i.\text{instructionAt}(pc_i) = \texttt{invokeinterface}\ m_i'' \wedge \\ S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: loc_i :: S_i' \wedge \\ H(loc_i).\text{class} = \sigma_i \wedge \\ m_{i-1} = \text{methodLookup}(m_i''.\text{id}, \sigma_i) \end{array}\right) \vee$$
$$\left(\begin{array}{l} m_i.\text{instructionAt}(pc_i) = \texttt{invokestatic}\ m_i'' \wedge \\ S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: S_i' \wedge \\ m_{i-1} = m_i'' \end{array}\right) \vee$$
$$\left(\begin{array}{l} m_i.\text{instructionAt}(pc_i) = \texttt{invokespecial}\ m_i'' \wedge \\ S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: loc_i :: S_i' \wedge \\ m_{i-1} = m_i'' \end{array}\right)$$

Intuitively the above definition states that a well formed frame is one in which the control stack is the result of method invocations and returns. These are indeed the frames of interest when considering a Carmel program, and the definition merely ensures that we need not consider any pathological control stacks.

The following Lemma shows that well-formedness of configurations is invariant under reduction:

Lemma 2 *If* $\langle K, H, SF\rangle$ *is well-formed and* $\langle K, H, SF\rangle \Rightarrow \langle K', H', SF'\rangle$ *then* $\langle K', H', SF'\rangle$ *is well-formed.*

Proof (sketch). By strong mathematical induction on the length of the call-stack.

The base case, for call stacks of length one, holds vacuously.

For the induction step, a case analysis is used on the instruction that is executed in the semantic step. There are only five interesting cases, since most instructions only modify the top element of the call stack. The `return` instruction on the other hand, removes the top element of the call-stack, thus *reducing* the size of the call-stack, and the case then follows immediately from the induction hypothesis. The remaining four cases (`invokevirtual`, `invokeinterface`, `invokestatic` and `invokespecial`) follow from inspection of the semantics. ■

Before we can state the subject reduction theorem, we need the following definition:

Definition 3 *Let* $SF = \langle m_1, pc_1, V_1, S_1\rangle :: \cdots :: \langle m_n, pc_n, V_n, S_n\rangle$ *and define the* address projection, π_{Adr}, *on* SF *as follows:*

$$\pi_{\mathrm{Adr}}(SF) = \{(m_i, pc_i) \mid i \in [1, n]\}$$

We are now finally in a position to prove the subject reduction Theorem:

Theorem 4 (Subject Reduction) *Let* $SF = \langle m, pc, V, S\rangle :: SF''$ *and* $I = m.\mathrm{instructionAt}(pc)$ *then if*

$$\frac{I = \mathit{instr}}{\langle K, H, SF\rangle \Rightarrow \langle K', H', SF'\rangle}$$

and

$$\beta_{\mathrm{Frame}}(\langle K, H, SF\rangle)\, \hat{\mathcal{R}}_{\mathrm{Frame}}\, (\hat{K}, \hat{H}, (\hat{L}, \hat{S}))$$

and

$$\forall (m, pc) \in \pi_{\mathrm{Adr}}(SF) : (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \mathit{instr}$$

and $\langle K, H, SF\rangle$ *is well-formed then*

$$\beta_{\mathrm{Frame}}(\langle K', H', SF'\rangle)\, \hat{\mathcal{R}}_{\mathrm{Frame}}\, (\hat{K}, \hat{H}, (\hat{L}, \hat{S}))$$

and $\langle K', H', SF'\rangle$ *is well-formed.*

Proof. By case inspection using Lemma 2 for the `return` instruction. A detailed proof can be found in Appendix A. ∎

6 Future Work

6.1 Extending the Instruction Set

One of the tasks in the immediate future is to finalise the formalisation details left out in this paper once the semantics is finalised. In the course of that work, the set of instructions supported will be gradually expanded to handle all of the Carmel instructions, including subroutines and support for exceptions. Based on the observations made in [5] regarding subroutines, it is most likely that support for them will not be added until late in the project.

Arrays play an important role in Java Card programming, cf. [6] and therefore warrant special attention. It is therefore important to study ways of increasing the precision of array analysis.

6.2 Formal Results

In the framework of Flow Logic, the proof of correctness is usually accompanied by a proof showing the existence of (best) acceptable analyses. Often this is proved by showing that the set of acceptable analyses is a Moore family.

6.3 Analysis Features

In order for the control flow analysis discussed in this paper to be useful for validating security and safety properties in Carmel, a number of additions and extensions will be needed, eg. exception analysis and ownership analysis. A number of such extensions are discussed in [4].

Apart from features and support discussed above, a number of "standard" additions and enhancements to the analysis may prove worthwhile to investigate, including reachability and escape analyses.

6.4 Implementation Issues

In order to obtain efficient implementations, it is important to study techniques for reducing the complexity of the analysis. We are currently investigating alternative and more efficient representations of the stack and local heap, similar in spirit to that of [13, 2], yet without transforming the language.

Another important way of reducing the complexity of the analysis would be to apply the *sharing* and *tiling* techniques possible for certain Flow Logic specifications described in [10].

References

[1] Zhiqun Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison Wesley, 2000.

[2] Marc Éluard and Thomas Jensen. Towards an operational semantics for Java Card byte code. SECSAFE-IRISA-001-0.1, February 2001.

[3] Stephen N. Freund and John C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'99*, pages 147–166, Denver, CO, USA, November 1999. ACM Press.

[4] René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-DRAFT. Forthcoming, 2002.

[5] Renaud Marlet. Proposition of a Hierarchy of Languages To Be Studied in SecSafe. SECSAFE-TL-004-1.0, December 2000.

[6] Renaud Marlet. Typical Code Patterns Found in Java Card Applets To Be Used as Targets for Program Analysis. SECSAFE-TL-003-1.0, December 2000.

[7] Renaud Marlet. Syntax of the JCVM Language To Be Studied in the SecSafe Project. SECSAFE-TL-005-1.7, May 2001.

[8] Flemming Nielson. Flow Logic. Web page: `http://www.daimi.au.dk/~fn/FlowLogic.html`.

[9] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[10] Flemming Nielson and Helmut Seidl. Control-Flow Analysis in Cubic Time. In *Proc. ESOP'01*, April 2001. SECSAFE-DAIMI-006-1.0 (preprint).

[11] Hanne Riis Nielson and Flemming Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. SECSAFE-DAIMI-001-1.0. Submitted for publication, February 2001.

[12] Igor Siveroni and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2, October 2001.

[13] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot — a Java Bytecode Optimization Framework. In *CASCON99*, September 1999.

[14] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. CC'92*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer Verlag, 1992.

A Proofs

Theorem 4 (Subject Reduction). *Let* $SF = \langle m, pc, V, S\rangle :: SF''$ *and* $I = m.\text{instructionAt}(pc)$ *then if*

$$\frac{I = \mathit{instr}}{\langle K, H, SF\rangle \Rightarrow \langle K', H', SF'\rangle}$$

and

$$\beta_{\text{Frame}}(\langle K, H, SF\rangle)\ \hat{\mathcal{R}}_{\text{Frame}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S}))$$

and

$$\forall (m, pc) \in \pi_{\text{Adr}}(SF) : (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \mathit{instr}$$

and $\langle K, H, SF\rangle$ *is well-formed then*

$$\beta_{\text{Frame}}(\langle K', H', SF'\rangle)\ \hat{\mathcal{R}}_{\text{Frame}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S}))$$

and $\langle K', H', SF'\rangle$ *is well-formed.*

Proof. First note that well-formedness follows from Lemma 2. We proceed by case inspection on the instruction.

Case (`push`): By assumption we have that

$$\frac{I = \texttt{push}\ \ t\ \ c}{\langle K, H, \langle m, pc, V, S\rangle :: SF\rangle \Rightarrow \langle K, H, \langle m, pc+1, V, c :: S\rangle :: SF\rangle}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{push}\ \ t\ \ c \tag{1}$$

and

$$\beta(\langle K, H, \langle m, pc, V, S\rangle :: SF\rangle)\ \hat{\mathcal{R}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \tag{2}$$

Now it follows from (1) that

$$\{c\} :: \hat{S}(m, pc) \quad \sqsubseteq \quad \hat{S}(m, pc+1) \tag{3}$$

$$\hat{L}(m, pc) \quad \sqsubseteq \quad \hat{L}(m, pc+1) \tag{4}$$

and from (2) we have that

$$\begin{aligned} \beta^H(K) &\sqsubseteq \hat{K} && (5)\\ \beta(H) &\sqsubseteq \hat{H} && (6)\\ \beta^H(V) &\sqsubseteq \hat{L}(m,pc) && (7)\\ \beta^H(S) &\sqsubseteq \hat{S}(m,pc) && (8) \end{aligned}$$

From (3) and (8) above it follows that

$$\begin{aligned} \beta^H(c :: S) &= \beta^H(c) :: \beta^H(S)\\ &= \{c\} :: \beta^H(S)\\ &\sqsubseteq \{c\} :: \hat{S}(m,pc)\\ &\sqsubseteq \hat{S}(m,pc+1) \end{aligned} \qquad (9)$$

By (4) and (7) we now obtain

$$\begin{aligned} \beta^H(V) &\sqsubseteq \hat{L}(m,pc)\\ &\sqsubseteq \hat{L}(m,pc+1) \end{aligned} \qquad (10)$$

The Theorem now follows from (5), (6), (9) and (10).

Case (`store`): By assumption we have

$$\frac{I = \texttt{store t i}}{\langle K, H, \langle m, pc, V, v :: S\rangle :: SF\rangle \;\Rightarrow\; \langle K, H, \langle m, pc+1, V[i \mapsto v], S\rangle :: SF\rangle}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{store}\ t\ i \qquad (11)$$

and

$$\beta(\langle K, H, \langle m, pc, V, v :: S\rangle :: SF\rangle)\ \hat{\mathcal{R}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \qquad (12)$$

From (12) it follows that

$$\begin{aligned} \beta^H(K) &\sqsubseteq \hat{K} && (13)\\ \beta(H) &\sqsubseteq \hat{H} && (14)\\ \beta(V) &\sqsubseteq \hat{L}(m,pc) && (15)\\ \beta^H(v :: S) &\sqsubseteq \hat{S}(m,pc) && (16) \end{aligned}$$

Now $\hat{S}(m,pc) = A :: X$ for some A and X, and (16) implies that

$$\begin{aligned} \beta^H(v) &\sqsubseteq A && (17)\\ \beta^H(S) &\sqsubseteq X && (18) \end{aligned}$$

Using (11) we have that

$$\begin{aligned} A &\sqsubseteq \hat{L}(m,pc+1)(i) && (19)\\ X &\sqsubseteq \hat{S}(m,pc+1) && (20) \end{aligned}$$

Combining (17) and (19) we obtain

$$\begin{aligned} \beta^H(v) &\sqsubseteq A\\ &\sqsubseteq \hat{L}(m,pc+1)(i) \end{aligned} \qquad (21)$$

and combining (18) and (20) it follows that

$$\begin{array}{rcl} \beta^H(S) & \sqsubseteq & X \\ & \sqsubseteq & \hat{S}(m, pc+1) \end{array} \tag{22}$$

From (11) and the definition of $\sqsubseteq_i$ we have that

$$\forall x \in \mathrm{dom}(\hat{L}(m, pc)) \setminus \{i\} : \hat{L}(m, pc)(x) \sqsubseteq \hat{L}(m, pc+1)(x) \tag{23}$$

Thus from (21) and (23) we have that

$$\beta^H(V[i \mapsto v]) \sqsubseteq \hat{L}(m, pc+1) \tag{24}$$

The Theorem now follows from (13), (14), (22) and (24).

Case (`getfield this`): By assumption we have

$$\frac{I = \texttt{getfield this } f}{\langle K, H, \langle m, pc, V, S\rangle :: SF\rangle \Rightarrow \langle K, H, \langle m, pc+1, V, v :: S\rangle :: SF\rangle}$$

where $loc = V[0]$, $o = H(loc)$ and $v = o.\mathrm{fieldValue}(f.\mathrm{id})$. We furthermore have by assumption that

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{getfield this } f \tag{25}$$

and

$$\beta(\langle K, H, \langle m, pc, V, S\rangle :: SF\rangle)\ \hat{\mathcal{R}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \tag{26}$$

From (26) it follows that

$$\begin{array}{rcll} \beta^H(K) & \sqsubseteq & \hat{K} & (27) \\ \beta(H) & \sqsubseteq & \hat{H} & (28) \\ \beta^H(V) & \sqsubseteq & \hat{L}(m, pc) & (29) \\ \beta^H(S) & \sqsubseteq & \hat{S}(m, pc) & (30) \end{array}$$

It follows directly from (25) and (29) that

$$\begin{array}{rcl} \beta^H(V) & \sqsubseteq & \hat{L}(m, pc) \\ & \sqsubseteq & \hat{L}(m, pc+1) \end{array} \tag{31}$$

Now assuming that $o.\mathrm{class} = \sigma$ it follows from (29) that

$$\begin{array}{rcl} \beta^H(V[0]) & = & \beta^H(loc) \\ & = & (\mathrm{Ref}\ \sigma) \\ & \sqsubseteq & \hat{L}(m, pc)(0) \end{array} \tag{32}$$

Then from (25) and (32) we have that

$$\hat{H}(\mathrm{Ref}\ \sigma)(f) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1) \tag{33}$$

Using (28), the fact that $\sigma = o.\mathrm{class} = H(loc).\mathrm{class}$ and the definition of $\beta(H)(\mathrm{Ref}\ \sigma)$ we arrive at

$$\begin{array}{rcl} \beta^H(H(loc)) & = & \beta^H(o) \\ & \sqsubseteq & \beta(H)(\mathrm{Ref}\ \sigma) \\ & \sqsubseteq & \hat{H}(\mathrm{Ref}\ \sigma) \end{array} \tag{34}$$

and thus

$$\beta(o.\text{fieldValue}(f.\text{id})) \sqsubseteq \hat{H}(\text{Ref } \sigma)(f) \tag{35}$$

and therefore

$$\begin{array}{rcll} \beta^H(v :: S) & = & \beta^H(v) :: \beta^H(S) & \\ & \sqsubseteq & \beta^H(v) :: \hat{S}(m, pc) & \\ & = & \beta(o.\text{fieldValue}(f.\text{id})) :: \hat{S}(m, pc) & (36) \\ & \sqsubseteq & \hat{H}(\text{Ref } \sigma)(f) :: \hat{S}(m, pc) & \\ & \sqsubseteq & \hat{S}(m, pc + 1) & \end{array}$$

The Theorem now follows from (27), (28), (31) and (36).

Case (`invokevirtual`): By assumption we have

$$\frac{I = \texttt{invokevirtual } m'}{\begin{array}{l}\langle K, H, \langle m, pc, V, v_1 :: \cdots :: v_n :: loc :: S\rangle :: SF\rangle \Rightarrow \\ \quad \langle K, H, \langle m_v, 1, V', \epsilon\rangle :: \langle m, pc, V, v_1 :: \cdots :: v_n :: loc :: S\rangle :: SF\rangle\end{array}}$$

where $n = |m'|$, $o = H(loc)$, $m_v = \text{methodLookup}(m'.\text{id}, o.\text{class})$ and $V' = loc :: v_1 :: \cdots :: v_n$. Furthermore, by assumption we also have

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{invokevirtual } m' \tag{37}$$

and

$$\beta(\langle K, H, \langle m, pc, V, v_1 :: \cdots :: v_n :: loc :: S\rangle :: SF\rangle)\ \hat{\mathcal{R}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \tag{38}$$

From (38) it immediately follows that

$$\begin{array}{rcll} \beta^H(K) & \sqsubseteq & \hat{K} & (39) \\ \beta(H) & \sqsubseteq & \hat{H} & (40) \\ \beta^H(V) & \sqsubseteq & \hat{L}(m, pc) & (41) \\ \beta^H(v_1 :: \cdots :: v_n :: loc :: S) & \sqsubseteq & \hat{S}(m, pc) & (42) \end{array}$$

From (37) we know that $\hat{S}(m, pc) = A_1 :: \cdots :: A_n :: B :: X$ for some $A_1, \ldots, A_N, B, X$ and thus (38) entails that

$$\begin{array}{rcll} \beta^H(v_i) & \sqsubseteq & A_i & (43) \\ \beta^H(loc) & \sqsubseteq & B & (44) \\ \beta^H(S) & \sqsubseteq & X & (45) \end{array}$$

Now, assuming that $o.\text{class} = \sigma$ we have that $\beta^H(loc) = (\text{Ref } \sigma)$ and combining (37) with (44) we obtain the following

$$\{(\text{Ref } \sigma)\} :: A_1 :: \cdots :: A_n \sqsubseteq \hat{L}(m_v, 1)[0..n] \tag{46}$$

because $m_v = \text{methodLookup}(m'.\text{id}, \sigma)$. From this it is clear that

$$\beta^H(V') \sqsubseteq \hat{L}(m_v, 1) \tag{47}$$

and

$$\beta^H(\epsilon) \sqsubseteq \hat{S}(m_v, 1) \tag{48}$$

The Theorem now follows from (39), (40), (47) and (48).

Case (`return`): By assumption we have

$$\frac{I = \texttt{return}\ t}{\begin{array}{c}\langle K, H, \langle m, pc, V, v :: S\rangle :: \langle m', pc', V', S'\rangle :: SF\rangle \Rightarrow \\ \langle K, H, \langle m', pc' + 1, V', v :: S'\rangle :: SF\rangle\end{array}}$$

It also follows from the assumptions that

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{return}\ t \tag{49}$$

and

$$\beta\langle K, H, \langle m, pc, V, v :: S\rangle :: \langle m', pc', V', S'\rangle :: SF\rangle\ \hat{\mathcal{R}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \tag{50}$$

and from this we immediately conclude that

$$\begin{aligned} \beta^H(K) &\sqsubseteq \hat{K} && (51)\\ \beta(H) &\sqsubseteq \hat{H} && (52)\\ \beta^H(V) &\sqsubseteq \hat{L}(m, pc) && (53)\\ \beta^H(v :: S) &\sqsubseteq \hat{S}(m, pc) && (54)\\ \beta^H(V') &\sqsubseteq \hat{L}(m', pc') && (55)\\ \beta^H(S') &\sqsubseteq \hat{S}(m', pc') && (56) \end{aligned}$$

From Lemma 2 it follows that the instruction at m'.instructionAt(pc') is one of either `invokevirtual`, `invokestatic` or `invokespecial` and thus:

$$\hat{L}(m', pc') \sqsubseteq \hat{L}(m', pc' + 1) \tag{57}$$

Combining this with (55) we arrive at

$$\beta^H(V') \sqsubseteq \hat{L}(m', pc' + 1) \tag{58}$$

From the above we now have three subcases to consider, depending on the instruction at m'.instructionAt(pc'). Assume now that

$$m'.\text{instructionAt}(pc') = \texttt{invokevirtual}\ m''$$

then Lemma 2 gives us that

$$m = \text{methodLookup}(m''.\text{id}, H(loc).\text{class}) \tag{59}$$

and

$$S' = v'_1 :: \cdots :: v'_n :: loc :: S'' \tag{60}$$

where $n = |m|$. From (56) we now know that $\hat{S}(m', pc') = A_1 :: \cdots :: A_n :: B :: X$ and therefore that

$$H(loc).\text{class} = \beta^H(loc) \sqsubseteq B \tag{61}$$

By assumption we have that

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m', pc') : \texttt{invokevirtual}\ m''$$

which combined with (59) and (61) gives us that $\hat{S}(m, \mathsf{END}_m) = T :: Y$ and that

$$T :: X \sqsubseteq \hat{S}(m', pc' + 1) \tag{62}$$

but from (56) we have that

$$\beta^H(S') \sqsubseteq X \tag{63}$$

and from (49) we have that $\hat{S}(m, pc) \sqsubseteq \hat{S}(m, \mathsf{END}_m)$ which implies that

$$\beta^H(v :: S) = \beta^H(v) :: \beta^H(S) \sqsubseteq \hat{S}(m, \mathsf{END}_m) \tag{64}$$

and therefore we know that

$$\beta^H(v) \sqsubseteq T \tag{65}$$

Combining the above we arrive at

$$\begin{array}{rcl} \beta^H(v :: S') & = & \beta^H(v) :: \beta^H(S') \\ & \sqsubseteq & T :: X \\ & \sqsubseteq & \hat{S}(m', pc' + 1) \end{array} \tag{66}$$

The two remaining subcases are similar.

The Theorem now follows from (51), (52), (58) and (66).

Case (`arrayload`): By assumption we have

$$\frac{I = \texttt{arrayload}\ t}{\langle K, H, \langle m, pc, V, i :: loc :: S\rangle :: SF\rangle \Rightarrow \langle K, H, \langle m, pc + 1, V, v :: S\rangle :: SF\rangle}$$

where $v = H(loc).\text{values}(i)$ and $H(loc).\text{elementType} = \tau$. It also follows from the assumptions that

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{arrayload}\ t \tag{67}$$

and

$$\beta\langle K, H, \langle m, pc, V, i :: loc :: S\rangle :: SF\rangle\ \hat{\mathcal{R}}\ (\hat{K}, \hat{H}, (\hat{L}, \hat{S})) \tag{68}$$

and from this we immediately conclude that

$$\begin{array}{rcll} \beta^H(K) & \sqsubseteq & \hat{K} & (69) \\ \beta(H) & \sqsubseteq & \hat{H} & (70) \\ \beta^H(V) & \sqsubseteq & \hat{L}(m, pc) & (71) \\ \beta^H(i :: loc :: S) & \sqsubseteq & \hat{S}(m, pc) & (72) \end{array}$$

Now it follows from (67) and (71) that

$$\begin{array}{rcl} \beta^H(V) & \sqsubseteq & \hat{L}(m, pc) \\ & \sqsubseteq & \hat{L}(m, pc + 1) \end{array} \tag{73}$$

From (67) we know that $\hat{S}(m, pc) = A :: B :: X$ for some A, B and X; now (72) implies that

$$\begin{array}{rcll} \beta^H(i) & \sqsubseteq & A & (74) \\ \beta^H(loc) & \sqsubseteq & B & (75) \\ \beta^H(S) & \sqsubseteq & X & (76) \end{array}$$

Since $\beta^H(loc) = (\text{Ref } (\texttt{array } \tau))$, eq. (75) implies $(\text{Ref } (\texttt{array } \tau)) \in B$ and thus from (67) we deduce that

$$\hat{H}(\text{Ref } (\texttt{array } \tau)) :: X \sqsubseteq \hat{S}(m, pc+1) \tag{77}$$

Furthermore we have that

$$\begin{array}{rcl} \beta^H(v) & = & \beta^H(H(loc).\text{values}(i)) \\ & \sqsubseteq & \beta^H(H(loc)) \\ & \sqsubseteq & \beta(H)(\text{Ref } (\texttt{array } \tau)) \\ & \sqsubseteq & \hat{H}(\text{Ref } (\texttt{array } \tau)) \end{array} \tag{78}$$

Finally from (76), (78) and (67) we have that

$$\begin{array}{rcl} \beta^H(v :: S) & = & \beta^H(v) :: \beta^H(S) \\ & \sqsubseteq & \hat{H}(\text{Ref } (\texttt{array } \tau)) :: X \\ & \sqsubseteq & \hat{S}(m, pc+1) \end{array} \tag{79}$$

The Theorem new follows from (69), (70), (73) and (79).

The remaining cases are similar. ■