INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A constraint-based algorithm for analysing memory usage on Java cards

Gerardo Schneider

N°????

Décembre 2004 - Updated: Mai 2005

Systèmes symboliques

Rapport de recherche

ISSN 0249-6399

A constraint-based algorithm for analysing memory usage on Java cards

Gerardo Schneider *

Systèmes symboliques
Projet Lande

Rapport de recherche n˚ ???? — Décembre 2004 - Updated: Mai 2005 — 25 pages

Abstract: We address in this paper the problem of statically determining whether a JavaCard applet may produce a memory overflow because of the dynamic instantiation of classes inside cycles. We provide a constraint-based algorithm which determines potential loops and (mutually) recursive methods. The algorithm operates on the byte-code of an applet. It is written as a set of rules –one for each byte-code instruction– which allows a compositional reasoning and it comprises both inter- and intra-procedural analysis. We aimed at an algorithm suitable to be fed into the proof assistant Coq in order to extract a certified memory usage analyser. We prove termination of the algorithm as well as its soundness and completeness with respect to an abstraction of the operational semantics of the language.

Key-words: Static analysis, memory analysis, constraint-based algorithm, Java card.

(Résumé : tsvp)

Supported by the *RNTL* French project, CASTLES (*Conception d'Analyses Statiques et de Tests pour le Logiciel Embarqué Sécurisé*).

* Gerardo.Schneider@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un algorithme à base de contraintes pour analyser l'utilisation de mémoire dans les cartes à puces Java

Résumé : Ce travail s'intéresse à déterminer statiquement si une application JavaCard peut produire un débordement de mémoire à cause de l'instantiation dynamique de classes dans les boucles. Nous définissons un algorithme à base de contraintes pour détecter les boucles et les méthodes mutuellement récursives. Notre algorithme opère sur des programmes «byte-code». Il est présenté comme un ensemble de règles –une pour chaque instruction byte-code– qui permet des raisonnement compositionels ; il consiste en une analyse intra- et inter-procédurale. Nous envisageons un algorithme qui puisse être décrit en Coq pour obtenir un analyseur d'utilisation de mémoire certifié. Nous en prouvons la terminaison ainsi que la correction et la complétude par rapport à une abstraction de la sémantique opérationnelle du langage.

Mots-clé : Analyse statique, analyse de mémoire, algorithme à base de contraintes, carte à puces Java

1 Introduction

Embedded systems, i.e. special-purpose computer systems built into larger devices, are widespread and can be found in simple gadgets like coffee machines and in more complex ones like mobile phones and smart cards. Programmable smart cards are small personal devices provided with a microprocessor capable of manipulating confidential data, allowing the owner of the card to have secure access to chosen applications. Applications, called applets, can be downloaded and executed in these small communicating devices, raising major security issues. Indeed, without appropriate security measures, a malicious applet could be installed in smart cards destroying data, disclosing private information over the network or performing non-authorised credit card transactions.

The massive use of such small communicating devices is imminent. Hence, it is essential that their platforms, as well as the applets that will run on them, can provide some minimal security guarantee in order to preserve confidentiality, integrity and availability of information. To guarantee availability of the services offered by small devices the management and control of resources (e.g. memory) is a crucial issue, mainly due to their limitation on size.

Currently, the maximum amount of memory for high-tech smart cards are 64K of EEPROM, 200K of ROM, 4K of RAM and a data bus of 32 bits. Memory for banking cards (and other low-end cards) are 16K, 16K, 1K and 8 bits respectively. Thus, memory is a very precious resource which must be manipulated carefully. For this reason, books on Java card technology provides advices to applet developers concerning memory management. We quote Chen: “Because memory in a smart card is very scarce, neither persistent nor transient objects should be created willy-nilly” [Che00, Section 4.4]. Chapter 13 of the same book provides programming tips on how to optimise applet creation considering the memory limitations: “You should also limit nested method invocations, which could easily lead to stack overflow. In particular, applets should not use recursive calls.” ([Che00, Section 13.3]). Later, on Section 13.4, it says: “An applet should always check that an object is created only once”. Even though the above tips are followed by industry when programming applets and are in general respected, mistakes regarding memory usage –like the instantiation of a class inside a loop– both accidentally or intentionally, may have non-desirable consequences. In fact, there is nothing in the standards which prevents a(n) (intentionally) badly written applet to allocate all persistent memory on a card. Hence, the detection of recursive methods and loops in byte-codes for embedded systems (smart cards in particular) is a crucial issue to take into account.

As far as we are aware there is no available tool for detecting the dynamic instantiation of classes inside cycles and/or recursive functions for Java smart cards. The byte-code verifier [Ler01, Ler02], for instance, does not verify properties related to memory usage. There are, undoubtedly, many ways of writing algorithms for detecting syntactic loops, mutually recursive methods (e.g., [ASU86]) and even for over-approximating the dynamic memory used for an assembler-like language. In this work, however, we propose a constraint-based algorithm, presented as a set of rules –one for each instruction– for solving the above-mentioned memory-related concerns. The formalism chosen was not governed by mere aesthetic reasons: it is compositional and it allows to obtain a certified analyser using the program-as-proof paradigm (i.e., following the Curry-Howard isomorphism) to extract a program directly from the proof of its correctness. One feature of our algorithm is that it works directly on the byte-code and there is no need to build extra data structures (e.g., a control-flow graph). Moreover, the byte-code considered is arbitrary, i.e. we do not have any assumption about the byte-code being produced by a “good” compiler. Our approach includes both intra- and inter-procedural analysis. Furthermore, the formalism used is compatible with existing works done at IRISA [CJPR04], which will allow to reuse the constraint solver, the fix-point algorithm and some of the lattice libraries "implemented" in Coq [Pic]. Our technique lies within the scope of *static analysis* [NNH99], which analyses some run-time properties of a program without executing it.

Our final goal is to obtain a *certified analyser*, using the extraction mechanism of the proof assistant Coq [BP04], for guaranteeing that no dynamic instantiation of classes are performed inside potential cycles and providing an over-approximation of the memory used by an applet. Such an analyser could be added as a functionality of the byte-code verifier. This paper is a first step in such direction.

The language being considered is JCVMLe [SH01], a *byte-code* language that models the Java Card Virtual Machine Language. It manipulates (dynamic) objects and arrays and besides the usual stack and register operations it comprises interesting features like virtual method calls, (mutually) recursive methods, (un)conditional jumps and subroutines. We assume there is no garbage collector, which is the case for Java Card upto version 2.1. Starting with Java Card 2.2 the machine includes a garbage collector which may be activated invoking an API function at the end of the execution of the applet, not during execution.

The paper is organised as follows. Section 2 briefly presents the language being considered while in Section 3 we present our algorithm. We prove, in Section 4, termination of our algorithm as well as its soundness and completeness with respect to an abstraction of the operational semantics of the language. Before concluding, in Section 5 we discuss how the algorithm presented may be improved in order to increase modularity.

2 The language

We consider in this paper the whole JCVMLe language except for subroutines calls and exception handling. It comprises, among others, the following kind of instructions:

- Stack manipulation: `push`, `pop`, `dup`, `dup2`, `swap`, `numop`;
- Local variables manipulation: `load`, `store`;
- Jump instructions: `if`, `goto`;
- Heap manipulation: `new`, `putfield`, `getfield`;
- Array instructions: `arraystore`, `arrayload`;
- Method calls and return: `invokevirtual`, `invokedefinite`, `invokeinterface`, `return`.

The whole instruction set, as well as an operational semantics of JCVMLe is given in [SH01].

A JCVMLe program P (which will be called in what follows a *byte-code program* or simply, a *program*) is a set of methods $m \in Method_P$ consisting of a numbered sequence of instructions. We write $InstrAt(m, pc) =$ `instr` to denote that the instruction at program line pc (usually called a *pc-number*) in method m is `instr`. Let $ProgCounter_P$ be the set of all the pc-numbers. We will usually omit the subscript P, being understood that the analysis depends on a given program P.

3 A constraint-based algorithm

We present in this section a constraint-based algorithm for detecting the occurrence of a `new` instruction inside potential cycles, due to intra-method loops and/or (mutually) recursive method call. The algorithm consists on three functions *Loop*, *Loop*' and *Rec* which computes the intra-method cycles, the methods called from intra-method cycles and the (mutually) recursive methods (as well as the methods reachable from those), respectively. With the information given by these functions, the main algorithm check whether a `new` instruction occurs inside a potential cycle. The main algorithm and all the above-mentioned functions are presented as a set of rules, each rule associating one or more constraints to each instruction of the given program. The solution of the set of constraints (which are of the form $\{F_1(X) \sqsubseteq X, \ldots F_n(X) \sqsubseteq X\}$) is the solution of a least fix-point operator F obtained from $F_1, \ldots F_n$. By a corollary of Tarski's Fix-point Theorem, this element may be obtained as the limit of the stabilising sequence $(F^n(\perp))_{n \in \mathbb{N}}$.

We will show first how to detect intra-method loops and (mutually) recursive methods, before showing our main algorithm.

3.1 Detection of loops

We will define two functions, *Loop* and *Loop*' for detecting cycles in a given program P: *Loop* will be defined intra-procedurally while *Loop*' will propagate the result given by *Loop* inter-procedurally.

Intra-procedural analysis. Let $\mathbb{Y}$ be defined as $\mathbb{Y} \stackrel{\text{def}}{=} \{\, Yes_{pc} \mid pc \in ProgCounter\}$. Given a program P, we define the following lattice:

$$\mathcal{L} = (Method \times ProgCounter \rightarrow \wp(ProgCounter \uplus \mathbb{Y}), \sqsubseteq)$$

where *Method* and *ProgCounter* are the *flat lattices* of method names and *pc*-numbers of P, respectively, and $\wp(ProgCounter \uplus \mathbb{Y})$ is the lattice of the powerset of *pc*-numbers and elements of $\mathbb{Y}$ ordered by the usual subset relation. The order of the above lattice is defined as follows: $f \sqsubseteq f'$ iff $f(m, pc) \sqsubseteq f'(m, pc)$, for all (m, pc).

We do not assume any structure on the byte-code program being considered, except that each `goto` is intra-method. Table 1 shows the rules (one for each instruction) used for computing the function *Loop*. Notice that the origin of a potential loop is always a `goto` "up".

Rule (1) corresponds to the `goto` "up" where F_1 is the following function:

$$F_1(L_{m,pc}, pc') = \begin{cases} L_{m,pc} \cup \{\, Yes_{pc}\} & \text{if } \{pc, pc'\} \subseteq L_{m,pc} \\ L_{m,pc} \cup \{pc, pc'\} & \text{otherwise} \end{cases} \tag{8}$$

Intuitively, F_1 detects whether $InstrAt(m, pc)$ has been visited at least twice from $InstrAt(m, pc')$, in which case *Yes* is added to *Loop*. Let $\mathbb{Y}_{<pc'}$ be the following set: $\mathbb{Y}_{<pc'} \stackrel{\text{def}}{=} \{\, Yes_{pc} \mid pc < pc'\}$. The case for a `goto` "down" is shown in rule (2) where F_2 is the function defined as follows:

$$F_2(L_{m,pc}, pc') = \begin{cases} L_{m,pc} \setminus \mathbb{Y}_{<pc'} & \text{if } \{pc, pc'\} \subseteq L_{m,pc} \\ (L_{m,pc} \setminus \mathbb{Y}) \cup \{pc, pc'\} & \text{otherwise} \end{cases} \tag{9}$$

where "\" is the usual set subtraction operator. F_2 propagates all the program counter numbers and all $Yes_{pc''}$ for *pc*" greater than *pc*', filtering all *pc*" smaller than *pc*'.

$$\frac{(m,pc) : \texttt{goto}\ pc' \qquad pc' \leq pc}{F_1(Loop(m,pc),pc') \sqsubseteq Loop(m,pc')} \quad (1)$$

$$\frac{(m,pc) : \texttt{goto}\ pc' \qquad pc' > pc}{F_2(Loop(m,pc),pc') \sqsubseteq Loop(m,pc')} \quad (2)$$

$$\frac{(m,pc) : \texttt{if}\ t\ op\ \texttt{goto}\ pc' \qquad pc' \leq pc}{\begin{array}{l} F_1(Loop(m,pc),pc') \sqsubseteq Loop(m,pc') \\ F_3(Loop(m,pc),pc') \sqsubseteq Loop(m,pc+1) \end{array}} \quad (3)$$

$$\frac{(m,pc) : \texttt{if}\ t\ op\ \texttt{goto}\ pc' \qquad pc' > pc}{\begin{array}{l} F_2(Loop(m,pc),pc') \sqsubseteq Loop(m,pc') \\ F_3(Loop(m,pc),pc') \sqsubseteq Loop(m,pc+1) \end{array}} \quad (4)$$

$$\frac{(m,pc) : \texttt{invokevirtual}\ m'}{Loop(m,pc) \sqsubseteq Loop(m,pc+1)} \quad (5)$$

$$\frac{(m,pc) : \texttt{return}}{\perp \sqsubseteq Loop(m,\mathrm{END}_m)} \quad (6)$$

$$\frac{(m,pc) : \texttt{instr}}{Loop(m,pc) \sqsubseteq Loop(m,pc+1)} \quad (7)$$

Table 1: Rules for *Loop*

```
30 if goto 50
31 goto 49      {30,31}
   ...
40 goto 60      {30,50,31,49,40}
   ...
49 if goto 60   {30,31,49}
50 goto 40      {30,50,31,49}
   ...
60 ...          {30,31,49,60,40}
```

(a)

```
20 ...          {30,50,31,41,40,70,20,Y70}
30 if goto 50   {30,50,31,41,40,70,20,Y70}
   ...          {30,31,50,41,40,51,70,20,Y70}
40 if goto 90   {30,31,50,41,40,51,70,20,Y70}
   ...          {30,31,41,40,50,51,70,20,Y70}
50 if goto 90   {30,31,41,40,50,51,70,20,Y70}
   ...          {30,31,41,40,50,51,70,20,Y70}
70 goto 20      {30,31,41,40,50,51,70,20,Y70}
   ...
90 ...          {30,31,40,90,41,50,51,70,20}
```

(b)

Figure 1: Example

Rule (3) defines *Loop* for the conditional jump, where F_1 is as before and F_3 is defined as follows:

$$F_3(L_{m,pc},pc') = \begin{cases} L_{m,pc} \setminus \mathbb{Y}_{<pc+1} & \text{if } \{pc,pc+1\} \subseteq L_{m,pc} \\ (L_{m,pc} \setminus \mathbb{Y}) \cup \{pc,pc+1\} & \text{otherwise} \end{cases} \tag{10}$$

F_3 is similar to F_2: it allows to propagate all the pc-numbers and only the $Yes_{pc''}$ such that pc" is greater than $pc+1$. This is needed since the `if` instruction is the condition for finishing a loop and clearly the next instruction is not inside such loop, without eliminating the case of an outer loop containing $InstrAt(m,pc+1)$. Rule (4) shows the case of the `if` "down". Rule (5) concerns virtual method invocation. Since we are considering here only intra-procedural cycles, the content of *Loop* is not propagated to method m'. Similar rules for `invokedefinite` and `invokeinterface` may be considered. The `return` instruction –rule (6)– does not propagate anything, since we are in an intra-method analysis. In rule (7), `instr` stands for any other instruction not defined by the rules (1)–(6); in this case the information at *Loop* is simply propagated to the next instruction.

We will usually write $Yes \in Loop(m,pc)$ whenever $\exists Yes_{pc'} \in Loop(m,pc)$. We define the following predicate: $Loop_{m,pc} \equiv Yes \in Loop(m,pc)$. We say that $InstrAt(m,pc)$ is in a loop iff $Loop_{m,pc}$.

$$\frac{(m,pc):\texttt{invokevirtual}\,m' \quad Loop_{m,pc}}{\begin{array}{c}Loop'(m,pc)\cup\{Yes\}\sqsubseteq Loop'(m',1)\\ Loop'(m,pc)\sqsubseteq Loop'(m,pc+1)\end{array}} \quad (11)$$

$$\frac{(m,pc):\texttt{invokevirtual}\,m' \quad \neg Loop_{m,pc}}{\begin{array}{c}Loop'(m,pc)\sqsubseteq Loop'(m',1)\\ Loop'(m,pc)\sqsubseteq Loop'(m,pc+1)\end{array}} \quad (12)$$

$$\frac{(m,pc):\texttt{instr}}{Loop'(m,pc)\sqsubseteq Loop'(m,pc+1)} \quad (13)$$

$$\frac{(m,pc):\texttt{return}}{\bot\sqsubseteq Loop'(m,\mathrm{END}_m)} \quad (14)$$

Table 2: Rules for *Loop*'

$$\frac{(m,pc):\texttt{invokevirtual}\,m' \quad m=m'}{\begin{array}{c}Rec(m,pc)\cup\{m,Yes\}\sqsubseteq Rec(m',1)\\ Rec(m,pc)\sqsubseteq Rec(m,pc+1)\end{array}} \quad (15)$$

$$\frac{(m,pc):\texttt{invokevirtual}\,m' \quad m\neq m'}{\begin{array}{c}F(Rec(m,pc),m')\sqsubseteq Rec(m',1)\\ Rec(m,pc)\sqsubseteq Rec(m,pc+1)\end{array}} \quad (16)$$

$$\frac{(m,pc):\texttt{return}}{Rec(m,pc)\sqsubseteq Rec(m,\mathrm{END}_m)} \quad (17)$$

$$\frac{(m,pc):\texttt{instr}}{Rec(m,pc)\sqsubseteq Rec(m,pc+1)} \quad (18)$$

Table 3: Rules for *Rec*

Example. In Fig. 1 we show part of two non so well-structured byte-code programs; the result of *Loop* is shown at the right part of each program. Program (a) contains no cycle, while in program (b) all the lines between 20 and 70 are inside a cycle originated at program line 70. □

Inter-procedural analysis. Given a program P, we define the following lattice $\mathcal{L}$:

$$\mathcal{L} = (Method \times ProgCounter \to \{\bot,\top\},\sqsubseteq),$$

where *Method* and *ProgCounter* are as before and $\{\bot,\top\}$ is the usual lattice with $\bot\sqsubseteq\top$. The order of the above lattice is defined as $f\sqsubseteq f'$ iff $f(m,pc)\sqsubseteq f'(m,pc)$, for all (m,pc). For convenience we will write *Yes* for $\top$.

The function *Loop*' captures all the instructions reachable from intra-procedural cycles through method calls, extending *Loop*. It is defined by the rules shown in Table 2. For sake of simplicity, the first constraint for rule (11) has been written $Loop'(m,pc)\cup\{Yes\}\sqsubseteq Loop'(m',1)$, but it must be understood as: $\forall m_i$ implementing m', $Loop'(m,pc)\cup\{Yes\}\sqsubseteq Loop'(m_i,1)$ and similarly for rule (12). The same remark holds for the `invokevirtual` rules of the function *Rec*.

3.2 Detection of (mutually) recursive methods

Given a program P, we define a lattice $\mathcal{L}$ as follows:

$$\mathcal{L} = (Method \times ProgCounter \to \wp(Method \uplus \{Yes\}),\sqsubseteq)$$

with the following order, $f\sqsubseteq f'$ iff $f(m,pc)\sqsubseteq f'(m,pc)$.

Rec takes values over the above lattice. The definition of *Rec* is given by the rules described in Table 3. The first rule corresponds to the case of a recursive method m. In this case, *Rec* at the first instruction of the method is marked as *Yes*, and its own method name is propagated. The application of the other rules will mark all the instructions of the method as *Yes*. Rule

(16) shows the case of a non self-invocation. The content of *Rec* is propagated to the next instruction inside the method, but to $InstrAt(m', 1)$ the information propagated is determined by a function $F : \wp(Method \uplus \{Yes\}) \times Method \longrightarrow \wp(Method \uplus \{Yes\})$ defined as follows:

$$F(R_{m,pc}, m') = \begin{cases} R_{m,pc} \cup \{m, Yes\} & \text{if } \{m'\} \in R_{m,pc} \\ R_{m,pc} \cup \{m\} & \text{if } \{m'\} \notin R_{m,pc} \end{cases}$$

At each method call (m, pc) : `invokevirtual` m', F adds m to the called method; if the called method name is already in $Rec(m, pc)$, then also *Yes* is added. Intuitively, F detects whether a given method has been visited more than once following the same "path". The other rules only propagate the result defined by the rules corresponding to `invokevirtual` (as before `instr` stands for any other instruction other of the defined by the rules (15)–(17)).

For a given method m and program counter pc, we define the predicate $Rec_{m,pc} \equiv Yes \in Rec(m, pc)$.

Remark. Notice that we are interested in knowing which methods may be executed an unknown number of times due to recursion. Thus, *Rec* detects not only all the (mutually) recursive methods but also all the methods which are called from any (mutually) recursive method. Indeed, the information in *Rec* allows us to know which methods are (mutually) recursive and which ones are reachable from the former: for any method m such that $Rec_{m,1}$, if $m \in Rec(m, 1)$, m is (mutually) recursive, otherwise m is reachable from a (mutually) recursive method.

3.3 Main algorithm

In this section we present the rules of our main algorithm, which detects the dynamic instantiation of classes. As before, we need to define the underlying lattice. Let $MPC \stackrel{\text{def}}{=} Method \times ProgCounter$; given a program P, we define a lattice $\mathcal{L}$ with the order relation defined similarly as for *Loop*. Let $\mathbb{W} = \{<!>_{(m,pc)} \mid (m, pc) \in MPC\}$; the lattice is:

$$\mathcal{L} = MPC \rightarrow \wp(MPC \uplus \mathbb{W}).$$

As for *Loop* we will use pc to refer to arbitrary members of *ProgCounter* and $<!>$ for arbitrary elements of $\mathbb{W}$.

The only instructions sensitive to our analysis are the ones which increases the heap, namely the creation of array objects and class instances (see the operational semantics in [SH01]). For simplification we consider here only one instruction `new`, but we mean both `new`(cl) and `new`(array a), where a is an array type. The rules in Table 4 define a context Γ, which is only incremented whenever $InstrAt(m, pc) =$ `new`(cl). $\Gamma(m, pc)$ will contain $<!>_{(m,pc)}$ if the current instruction is inside a potential loop and/or in a (mutually) recursive method and it will contain pc otherwise. We write $Cycle_{m,pc}$ as a shortcut for $Loop_{m,pc} \vee Loop'_{m,pc} \vee Rec_{m,pc}$. For the other rules, the content of the context Γ is propagated, without any change.

4 Theoretical properties

4.1 Termination

One of the crucial properties for proving termination of our analyser is the *ascending chain condition*. The property is trivially satisfied by all our lattices since their width and height are

$$\frac{(m,pc):\texttt{new}(cl) \qquad Cycle_{m,pc}}{\Gamma(m,pc)\cup\{<!>_{(m,pc)}\}\sqsubseteq\Gamma(m,pc+1)} \qquad (19)$$

$$\frac{(m,pc):\texttt{new}(cl) \qquad \neg Cycle_{m,pc}}{\Gamma(m,pc)\cup\{(m,pc)\}\sqsubseteq\Gamma(m,pc+1)} \qquad (20)$$

$$\frac{(m,pc):\texttt{invokevirtual}\ m'}{\begin{array}{l}\Gamma(m,pc)\sqsubseteq\Gamma(m',1)\\ \Gamma(m',\mathrm{END}_{m'})\sqsubseteq\Gamma(m,pc+1)\end{array}} \qquad (21)$$

$$\frac{(m,pc):\texttt{return}}{\Gamma(m,pc)\sqsubseteq\Gamma(m,\mathrm{END}_m)} \qquad (22)$$

$$\frac{(m,pc):\texttt{if}\ t\ op\ \texttt{goto}\ pc'}{\begin{array}{l}\Gamma(m,pc)\sqsubseteq\Gamma(m,pc+1)\\ \Gamma(m,pc)\sqsubseteq\Gamma(m,pc')\end{array}} \qquad (23)$$

$$\frac{(m,pc):\texttt{goto}\ pc'}{\Gamma(m,pc)\sqsubseteq\Gamma(m,pc')} \qquad (24)$$

$$\frac{(m,pc):\texttt{instr}}{\Gamma(m,pc)\sqsubseteq\Gamma(m,pc+1)} \qquad (25)$$

Table 4: Rules for the main algorithm

finite. The algorithm reduces to the problem of solving a set of constraints over a lattice $\mathcal{L}$, which can be transformed into the computation of a fix-point of a monotone function over $\mathcal{L}$. Termination follows from the proof of the termination of the fix-point computation for *Loop*, *Rec* and the algorithm itself.

The following lemma establishes the monotonicity of functions F_1, F_2, F_3 and F used in the definition of *Loop* and *Rec*.

Lemma 1 *The functions F_1, F_2, F_3 and F used in the definition of Loop and Rec are monotone.*

Proof. See Lemmas 10–13 in Appendix A. □

It is well known (see for instance [NNH99]) that the solution of a set of constraints of the form $\{F_1(X) \sqsubseteq X, \ldots F_n(X) \sqsubseteq X\}$, where each F_i is monotone, is the solution of a least fix-point operator F obtained from $F_1, \ldots F_n$. Moreover, by a corollary of Tarski's Fix-point Theorem, this element may be obtained as the limit of the stabilising sequence $(F^n(\bot))_{n\in\mathbb{N}}$. As a corollary of the previous lemma we have the following result.

Corollary 2 *The computation of the least fix-point corresponding to the functions Loop, Loop' and Rec always terminate.* □

Termination of the algorithm follows from termination of *Loop*, *Loop*', *Rec* and from the "form" of the constraints. We have then:

Theorem 3 *The algorithm given by rules (19)–(25) by computing its corrresponding least fix-point, always terminate.* □

4.2 Soundness and Completeness

The semantics of a program P, $[\![P]\!]$ is the set of all the possible traces issued from the operational semantics. We prove soundness and completeness of the functions introduced in Section 3 w.r.t. an appropriate abstraction of the operational semantics. For the inter-procedural analysis (*Rec* and *Loop*') we consider the usual method-call graph, which is an abstraction of the control-flow graph $\mathcal{G}(P)$ of the program P, while for the intra-procedural case (*Loop*) we consider $\mathcal{G}_m(P)$, which is a subgraph of $\mathcal{G}(P)$, restricted to the given method m. The trace obtained from a traversal of the above graph without testing any jump nor branching conditions, is an *abstract trace* of P; let $[\![\widehat{P}]\!]$ denote the set of all the abstract traces of program P. In what follows we will use the fact that $[\![P]\!] \subseteq [\![\widehat{P}]\!]$ but we will not prove it (see for instance [NN99, DS98] and references therein).

4.2.1 *Loop*.

Given a program P, its *control-flow graph* $\mathcal{G}(P)$, is a 5-tuple $(\mathcal{S}, \mathcal{A}, \rightarrow, s_0, F)$, where

- $\mathcal{S}$ is a set of *nodes* (or *program states*), ranging over pairs of $\mathit{Method} \times \mathit{ProgCounter}$;
- $\mathcal{A}$ is a set of *actions*, ranging over byte-code instructions;
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a set of *labelled transitions*, which models the flow of control;
- s_0 is the initial state;
- F is a set of final states.

Whenever understood from the context we will write $\mathcal{G}$ instead of $\mathcal{G}(P)$. As usual, we will write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \rightarrow$. We say that s is the *predecessor* of s' if for some a, $s \xrightarrow{a} s'$ (similar for *successor*). The set of all the successors of a state s is given by the transitive closure of $\rightarrow$; we say that a state s' is *reachable* from s iff s' is a successor of s and we write $s' \in \mathit{Reach}^+(s)$. More formally[1],

$$\begin{aligned}
&\mathit{Reach}((m, pc) : \texttt{goto}\ pc') = (m, pc')\\
&\mathit{Reach}((m, pc) : \texttt{if}\ c\ \texttt{goto}\ pc') = \{(m, pc+1), (m, pc')\}\\
&\mathit{Reach}((m, pc) : \texttt{invokevirtual}\ m') = \{(m, pc+1), (m', 1)\}\\
&\mathit{Reach}((m, pc) : \texttt{return}) = (m, \mathrm{END}_m)\\
&\mathit{Reach}((m, \mathrm{END}_m)) = \{(m', pc+1) \mid \mathit{InstrAt}(m', pc) = \texttt{invokevirtual}\ m\}\\
&\mathit{Reach}((m, pc) : \texttt{instr}) = (m, pc+1)
\end{aligned}$$

Where `instr` is any other instruction different from `invokevirtual`, `goto`, `if`, and `return`. The transitive closure of the above relation is denoted Reach^+.

We introduce a *satisfaction* relation: $\mathcal{G} \models \phi$ iff $\mathcal{G}$ satisfies the property ϕ.

For our purposes, it is convenient to define the control-flow graph of each method independently, which may be obtained from the graph $\mathcal{G}$ as a subgraph, cutting all the transitions issued from `invokevirtual` instructions. For each method m, we will denote this graph by $\mathcal{G}_m$, which will be called the *m-control-flow graph.* When considering the graph $\mathcal{G}_m$, *Reach* is as before but for `invokevirtual`: $\mathit{Reach}((m, pc) : \texttt{invokevirtual}\ m') = (m, pc+1)$; moreover, there is no rule $\mathit{Reach}((m, \mathrm{END}_m))$. We define the following:

$$\begin{gathered}
\mathcal{G}_m \models \mathit{SynC}(m, pc)\\
\text{iff}\\
\mathcal{G}_m \models \exists pc' \cdot ((m, pc') \in \mathit{Reach}^+(m, pc) \wedge (m, pc) \in \mathit{Reach}^+(m, pc')).
\end{gathered}$$

Thus, the predicate *SynC* determines whether a given instruction is in a *syntactic cycle*. Notice that this predicate only characterises *intra-method* cycles. The following theorem establishes that the predicate *Loop* characterises exactly all the syntactic cycles of a method[2].

Theorem 4 (Soundness and Completeness of *Loop*) $\mathit{Loop}_{m,pc}$ *iff* $\mathcal{G}_m \models \mathit{SynC}(m, pc)$. □

[1] We will write $\mathit{Reach}((m, pc) : \texttt{goto}\ pc') = (m, pc')$ as a short for "If $s = (m, pc) \in \mathcal{S}$ and it exists $s' \in \mathcal{S}$ such that $s \xrightarrow{a} s'$, with $a = \texttt{goto}\ pc'$, then $\mathit{Reach}(m, pc) = (m, pc')$".

[2] See Lemmas 14 and 15 in Appendix A.

4.2.2 *Loop*'.

The following predicate, *SynCReach* determines whether a given instruction is reachable from a method invocation inside a syntactic cycle:

$$\mathcal{G} \models \mathit{SynCReach}(m, pc)$$
$$\text{iff}$$
$$\text{It exists } m',\ \mathcal{G}_{m'} \models \mathit{SynC}(m', pc') \text{ and } \mathcal{G} \models (m, pc) \in \mathit{Reach}^{+}(m', pc').$$

Loop' characterises exactly all instructions reachable from a cycle as expressed in the following theorem[3].

Theorem 5 (Soundness and Completeness of *Loop*') $\mathit{Loop}'_{m,pc}$ *iff* $\mathcal{G} \models \mathit{SynCReach}(m, pc)$. □

4.2.3 *Rec*.

Given a program P, its *method-call graph* $\mathcal{M}(P)$, is a 3-tuple $(\mathcal{N}, \rightarrow, m_0)$, where

- $\mathcal{N}$ is a set of *nodes* ranging over *Method* names;
- $\rightarrow \subseteq \mathcal{N} \times \mathcal{N}$ is a set of *transitions*;
- m_0 is the initial state.

Notice that the transitions in the above graph are not labelled and that there are no final states. Indeed, $\mathcal{M}(P)$ models the *method call* relationship: $m \rightarrow m'$ if and only if $\exists pc \cdot \mathit{InstrAt}(m, pc) =$ `invokevirtual` m'. Whenever understood from the context we will write $\mathcal{M}$ instead of $\mathcal{M}(P)$. Given the method-call graph, we say that a method m' is *reachable* from m if and only if there is a path in the graph from m to m'. More formally, $\mathit{Reach} = \{(m, m') \mid m \rightarrow m'\}$, and Reach^{+} is the transitive closure of the above relation. We say that $m' \in \mathit{Reach}^{+}(m)$ iff $\mathit{Reach}^{+}(m, m')$. Given the graph of method calls, we define a relation $\mathit{MutRec} \subseteq \mathit{Method} \times \mathit{Method}$ as $\mathit{MutRec} = \{(m, m') \mid m' \in \mathit{Reach}^{+}(m) \wedge m \in \mathit{Reach}^{+}(m')\}$. Hence, the following predicate determines whether a given method m is (mutually) recursive:

$$\mathit{MutRec}(m) \equiv \exists m' \cdot m' \in \mathit{Reach}^{+}(m) \wedge m \in \mathit{Reach}^{+}(m').$$

The following predicate characterises not only the (mutually) reachable methods but also the methods reachable from those:

$$\mathit{MutRecReach}(m) \equiv \exists m' \cdot \mathit{MutRec}(m') \wedge m \in \mathit{Reach}^{+}(m').$$

We introduce a *satisfaction* relation: $\mathcal{M} \models \phi$ iff $\mathcal{M}$ satisfies the property ϕ. We state now soundness and completeness of *Rec*[4].

Theorem 6 (Soundness and Completeness of *Rec*) $\mathit{Rec}_{m,pc}$ *iff* $\mathcal{M}(P) \models \mathit{MutRecReach}(m)$. □

[3] See Lemmas 27 and 28 in Appendix A.
[4] See Lemmas 29 and 30 in Appendix A.

4.2.4 Main algorithm.

We prove now the soundness of our algorithm w.r.t. to our abstraction.

Theorem 7 (Soundness of the algorithm) *If* $<!>_{(m,pc)} \in \Gamma$ *then* $I = InstrAt(m, pc) =$ `new`(cl) *and* I *occurs in a syntactic cycle and/or in a mutually recursive method. Furthermore, if* $(m, pc) \in \Gamma$ *then* $I = InstrAt(m, pc) =$ `new`(cl) *and* I *is not in a syntactic cycle nor in a mutually recursive method.*

Proof. Notice that $<!>_{(m,pc)}$ and (m, pc) are introduced into Γ only by rules (19) and (20), given that $InstrAt(m, pc) =$ `new`(cl), and whenever $Loop_{m,pc} \vee Loop'_{m,pc} \vee Rec_{m,pc}$ and its negation, respectively. The other rules only propagate the result according to the definition of *Reach*. Thus, the result follows from soundness of *Loop*, *Loop*' and *Rec*. □

We can also prove that our algorithm is complete w.r.t. the abstraction.

Theorem 8 (Completeness of the algorithm) *If* I *occurs in a syntactic cycle and/or in a mutually recursive method and* $I = InstrAt(m, pc) =$ `new`(cl)*, then* $<!>_{(m,pc)} \in \Gamma$*. On the other hand, if* $I = InstrAt(m, pc) =$ `new`(cl) *and* I *is not in a syntactic cycle nor in a mutually recursive method then* $(m, pc) \in \Gamma$*.*

Proof. The Theorem follows from the completeness of *Loop*, *Loop*' and *Rec* and by rules (19)–(20) which only add $<!>_{(m,pc)}$ to Γ whenever $InstrAt(m, pc) =$ `new`(cl) inside a cycle and $(m, pc) \in \Gamma$ if such instruction is not in a cycle. □Notice that the above soundness and completeness result are with respect to an abstraction, which is the identification of `new` instructions inside *syntactic* cycles. However, the real interesting conclusion is:

Corollary 9 *If* $(m, pc) \in \Gamma$ *then for any real execution of the applet,* $InstrAt(m, pc)$ *will be executed a finite number of times.* □

Our algorithm may be easily refined to give an upper bound of the memory used by an applet if no `new` instruction occurs inside a loop.

5 Improving Modularity

Our algorithm is not completely modular in the sense that the parallel composition of two applets may introduce new cycles not existing before the composition. In such case *Rec* should be recomputed again to guarantee that no `new` exists inside the new generated cycle. However, it could be possible to avoid computing a global fix-point (for detecting mutually recursive methods) if we keep relevant information regarding the methods of one applet called by methods of the other (and vice-versa).

Example. Let us consider the call graph of two applets A and A' in Fig. 2-(a). The shaded part is an inner loop. From the graph of A we can conclude that m_0 and m_2 are not in cycle and that from m_0 a method is called "outside" A (it's a *pending* call, depicted as a dashed arrow), while m_1 and m_3 are mutually recursive with a pending call inside the recursion. On the other hand, there is no mutually recursive methods in A' and there is only a pending call in method m'_1. In Fig. 2-(b) we can see that after the parallel composition a new cycle is introduced in the graph due to some of the pending calls, making m_0, m_2, m'_0 and m'_1 mutually recursive and m'_2 reachable from mutually recursive methods. □

We present now the sketch of an algorithm which avoids, in some cases, recomputing the global fix-point after composing two (or more) applets.

We assume that a preprocessing allows us to classify the `invokevirtual` instructions into *static* and *pending* calls if the method called may be statically identified or not, respectively. Let $Pred(m)$ ($Succ(m)$) be the set of all the *predecessors* (*successors*) –w.r.t. the call graph– of the method m; for M a set of methods, $Pred(M) = \cup_{m \in M} Pred(m)$. Let $Pend(A)$ be the set of all the methods of applet A containing a pending call. The algorithm is as follows:

1. Analyse A (compute *Loop*, *Loop*' and *Rec*);
2. Compute $Pend(A)$;
3. Compute $Pred(Pend(A))$;
4. Repeat the above steps for applet A';
5. Check whether $\exists m \in Pend(A) \;.\; \exists m' \in Pend(A') \;.\; (InstrAt(m, pc) = \texttt{invokevirtual}\ m'_? \wedge InstrAt(m', pc') = \texttt{invokevirtual}\ m_? \wedge m'_? \in Pred(m') \cup \{m'\} \wedge m_? \in Pred(m) \cup \{m\})$. If it is the case, then all the methods in $(Pred(m) \cap Succ(m_?)) \cup (Pred(m') \cap Succ(m'_?))$ are mutually recursive.

Remark. Notice that in the previous algorithm for obtaining *Pred* we may need to compute a fix-point, however, this may be avoided if we make a preprocessing distinguishing between the pending calls inside a loop (inner cycle or mutually recursive methods) and those that are outside loops. The pending call in method m_1 (see Fig. 2-(a)) is inside a mutually recursive method, hence we should not need to compute $Pred(m_1)$ –after "instantiating" the pending call, the called method will be automatically considered as (reachable from a) mutually recursive method. Only $Pred(m_0)$ and $Pred(m'_1)$ have to be computed.

Observe that it is not always possible to compute only local fix-points. Indeed, in the case of a "zig-zag" of calling methods between the two applets, then we need to compute a global fix-point. We conjecture, however, that this global fix-point would not need to be computed on the whole set of methods.

6 Final Discussion

This work is a first step in the construction of a certified analyser for estimating memory usage on Java cards. We have given a constraint-based algorithm which detects all the potential loops and (mutually) recursive methods, in order to determine the existence of dynamic instantiation of classes inside cycles. We have proved its soundness and completeness w.r.t. an abstraction of the operational semantics of the language being considered. Our abstraction is conservative and correct w.r.t. a run-time execution of the program, in the sense that if a run-time cycle exists, then such cycle is detected by our algorithm and if our analysis gives as an answer that no `new` instruction is inside a cycle, then it is indeed the case. Besides the advantages mentioned in the introduction, the modularity of our algorithm allows the analyser itself, as well

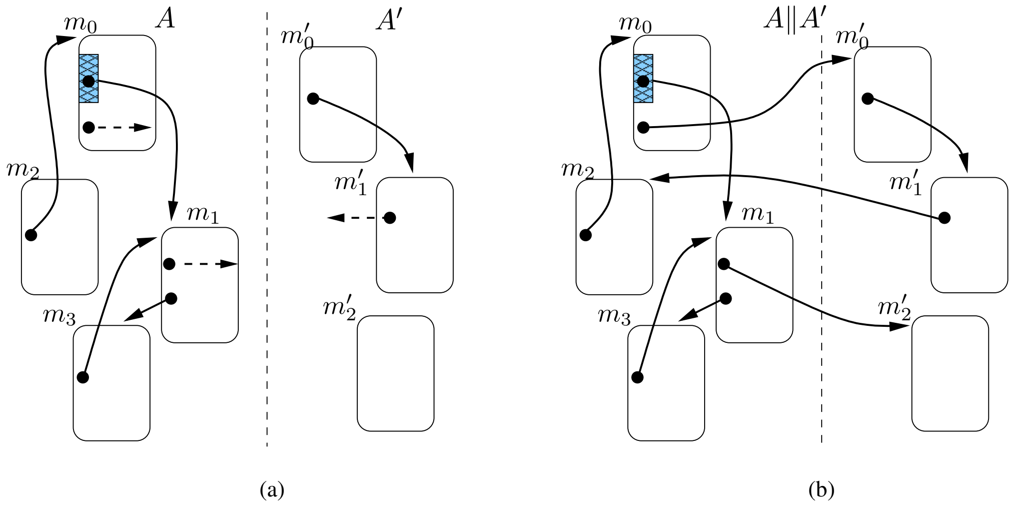

Figure 2: Example: (a) Two applets with pending calls; (b) After composition.

as the predicates (functions) *Loop* and *Rec*, to be reused by other constraint-based analysers as predicates (programs) which have been proved correct.

Regarding complexity issues, a complex method may have up-to 50 branching instructions and even considering that we may store the *pc*-numbers in 1 byte, we largely pass the barrier of 4 Kb of RAM (in the worst case we might need 40 Kb for computing *Loop*). This makes our analyser infeasible to be on-card. We believe, that many optimisations may be done to improve the RAM used by our algorithm in order to obtain an on-card analyser, increasing, however, time processing. The current version should be seen as an off-card tool for applet developers and to check applet provided by third part suppliers.

We have defined two predicates, *Loop* and *Loop*' for detecting cycles. Notice that we could have defined only one predicate *Loop* just changing the rule of `invokevirtual` to:

$$\frac{(m, pc) : \texttt{invokevirtual}\ m'}{\begin{array}{l} \mathit{Loop}(m, pc) \sqsubseteq \mathit{Loop}(m', 1) \\ \mathit{Loop}(m, pc) \sqsubseteq \mathit{Loop}(m, pc + 1) \end{array}}$$

In this case we would not need the definition of *Loop*'. Our choice is, however, not arbitrary and it is motivated by modularity and memory concerns. Computing the intra-procedural cycles first allows to have a local reasoning which may be done once and for all.

Possible Improvements. In what follows we mention some possible improvements on the current algorithm and in the subsection "Current and Future Work" below we sketch an alternative algorithm.

For simplicity we have not made any difference between transient and not transient objects to avoid a proliferation of rules, although this is clearly possible in our approach, obtaining then a discriminating memory estimation for the EEPROM and the RAM. Currently, our algorithm gives a program point (m, pc) only once if the instruction (m, pc) : `new` is not in a loop nor in a mutually recursive method. That means that if m is invoked, say 10 times, from different points in a program, the algorithm will give only one occurrence of (m, pc) in Γ. The algorithm may be easily refined in order to give a more accurate answer in the non-loop cases, being able to exactly give the memory used.

Some improvements may be done to improve space-complexity. The underlying lattice of *Loop*, for instance, may be simplified considering only *pc*-numbers and excluding the *Yes* elements. In this case the proof in Coq would be simpler and the fix-point computation will converge faster but an extra pass would be needed to "annotate" the cycles.

Another simpler alternative algorithm for detecting loops could be obtained if we consider that the byte-code is produced by a good compiler. This would give a more efficient, but less precise algorithm, probably missing real cycles. On the other hand the correctness of the approach would be maintained: if an instruction is said to be not in a cycle, then it would be indeed the case.

Complexity Issues. Let n be the number of *pc*-numbers, N_U the number of unconditional and N_C of conditional jump instructions of a method m of a given program P; let B be number of bytes used for representing a *pc*-number. An upper bound of the auxiliary memory used by *Loop* will be (in the worst case) $O((3N_U + 4N_C) \times B \times N)$: for each unconditional jump instruction we add 2 *pc*-numbers and one *Yes* if such instruction generates a cycle while for each conditional jump we add 3 *pc*-numbers and one *Yes* if such instruction generates a cycle. This information is propagated along the method, and in the worst case to all of its instructions. For *Rec* the auxiliary memory used is definitely less, since we only propagate method's names, which might be associated to each method without needing to propagate them to every method line. The efficiency of our algorithm (on time) strongly depends on the efficiency of the constraint solver, but we believe that in principle each fix-point computation converges in at most three iterations. Moreover, the order in which the different predicates are applied may drastically improve the performance of the analyser.

Related Work. Besides the logical approach, the use of type systems [Car97] is probably the most successful tool for guaranteeing that well typed programs run within stated space-bounds. Previous work along these lines had put the stress on the definition of typed assembly languages (e.g. [AC03, VC04]) or in the definition of type systems for functional languages [AH02, Hof00, HP99]. In [AC03] the authors present a first-order linearly typed assembly language which allows the safe reuse of heap space for elements of different types. The idea is to design a family of assembly languages which have high-level typing features (e.g. the use of a special *diamond* resource type) which are used to express resource bound constraints. Closely related to the previous-mentioned paper, [VC04] describes a type theory for certified code, in which type safety guarantees cooperation with a mechanism to limit the CPU usage of untrusted code. Another recent work is [ACGZJ04] where the resource bounds problem is studied in a simple stack machine. The authors show how to perform type, size and termination verifications at the level of the byte-code. In [NN99] it is shown how to import ideas from data flow analysis, using abstract interpretation, into inter-procedural control flow analysis. Even though the motivations and the techniques are not the same, it is worth mentioning the work done in the Mobile Resource Guarantees project [MRG], which applies ideas from proof-carrying code for solving the problem of resource certification for mobile code.

Our approach is inspired by the work in [CJPR04] where a similar technique as the one used here has been applied with success for extracting a data flow analyser for Java card byte-code programs. Our work is, undoubtedly, related to well-known works on compiler construction [ASU86] and on program analysis [NNH99]. Finally, Leroy's papers [Ler01, Ler02] are undoubtedly important contributions to the understanding of byte-code verification, although in such papers no solution to the problem addressed here is presented (see also references therein for more related work).

Current and Future Work. Since our ultimate goal is to produce an efficient certified analyser for estimating memory usage on Java card we are trying to find a good compromise between an easy-to-prove algorithm and an efficient one. We are currently working on an alternative algorithm with simpler rules but provably less efficient than the one presented in this work. To compute *Loop* we have chosen here to add to the context the *pc*-numbers of source and target of any jump; also *Yes* is added, in order to determine whether a given instruction is in a loop. A different algorithm may be obtained adding to the context the *pc*-numbers of the instructions already visited without adding *Yes*, simplifying then the underlying lattice but obliging a post-processing. We have not convincing argument for the choice made in this paper, but we believe it is more efficient than keeping track of all the visited instructions. Besides the rules corresponding to such an alternative algorithm being simpler, probably the reutilisation of previous works (e.g. [Pic] and [CJPR04]), will be higher. The proof technique used in such paper is based on *subject reduction*: if a prefix of a trace (run of the program) satisfies a predicate (e.g, *Loop*) then any extension of such trace must satisfy the predicate. Clearly, we need to consider a different approach to prove correctness, since in [CJPR04] the proof is conducted considering a *partial* semantics in contrast with the *maximal* semantics used in this work: the semantics of a program is given by the set of all the *total* traces.

In this paper we have given only a handwritten proof of termination, soundness and completeness. We intend to prove the correctness of our algorithm in the constructive logic of Coq and to use its extraction mechanism to automatically obtain a certified analyser. Once both implementations completed (this work as well as the algorithm discussed above), we intend to compare facility of proof and efficiency.

An extension of our constraint-based algorithm for treating exceptions is currently under study.

Acknowledgements. We are indebted to Pablo Giambiagi for pointing out a bug in an early version of the *Loop* predicate and to Thomas Jensen and David Pichardie for insightful discussions about the "good" way of presenting the rules in order to be suitable for Coq. Patrick Bernard, Eduardo Giménez, Arnaud Gotlieb and Jean–Louis Lanet have provided us with useful information on the Java card technology.

References

[AC03] D. Aspinall and A. Compagnoni. Heap-bounded assembly language. *J. Autom. Reason.*, 31(3-4):261–302, 2003.

[ACGZJ04] R.M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. Research report 17-2004, LIF, Marseille, France, 2004.

[AH02] D. Aspinall and M. Hofmann. Another type system for in-place update. In *ESOP*, volume 2305 of *LNCS*, pages 36–52, 2002.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BP04] Y. Bertot and P.Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in TCS. Springer Verlag, 2004.

[Car97] L. Cardelli. *Type Systems*, chapter 103. Handbook of Computer Science and Engineering. CRC Press, 1997.

[Che00] Z. Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Java series. Addison Wesley, 2000.

[CJPR04] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *ESOP*, LNCS, pages 385–400, 2004.

[DS98] B. Steffen D.A. Schmidt. Program analysis as model checking of abstract interpretations. In *SAS*, LNCS, pages 351–380, 1998.

[Hof00] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[HP99] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

[Ler01] X. Leroy. Java bytecode verification: an overview. In *CAV'01*, volume 2102 of *LNCS*, pages 265–285. Springer-Verlag, 2001.

[Ler02] X. Leroy. Bytecode verification on java smart cards. *Softw. Pract. Exper.*, 32(4):319–340, 2002.

[MRG] MRG. Mobile Resource Guarantees project. See `http://groups.inf.ed.ac.uk/mrg/`.

[NN99] F. Nielson and H.R. Nielson. Interprocedural control flow analysis. In *European Symposium on Programming*, pages 20–39, 1999.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[Pic] D. Pichardie. Coq sources of the development corresponding to the paper [CJPR04]. See `http://www.irisa.fr/lande/pichardie/CarmelCoq/`.

[SH01] I. Siveroni and C. Hankin. A proposal for the JCVMLe operational semantics. Research report SECSAFE-ICSTM-001-2.2, October 2001.

[VC04] J.C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, CMU, February 2004.

A Auxiliary Lemmas

A.1 Monotonicity

Lemma 10 *F_1 is monotonic.*

Proof. Remember that $(m, pc_m, A) \sqsubseteq_{\mathcal{L}} (n, pc_n, B)$ if and only if $m = n$, $pc_m = pc_n$ and $A \subseteq B$, where $A, B \in \wp(\mathit{ProgCounter} \uplus \mathbb{Y})$. F_1 is the function $F_1 : \wp(\mathit{ProgCounter} \uplus \mathbb{Y}) \times \mathit{ProgCounter} \rightarrow \wp(\mathit{ProgCounter} \uplus \mathbb{Y})$ defined in Eq. (8). Since $m = n$ and $pc_m = pc_n$, we will prove that if $(A, pc') \sqsubseteq (B, pc')$ then $F_1(A, pc') \sqsubseteq F_1(B, pc')$, or what is the same, we will prove that $a \in F_1(A, pc')$ implies $a \in F_1(B, pc')$, assuming $(A, pc') \sqsubseteq (B, pc')$. We make a case-analysis:

$\{pc, pc'\} \subseteq A$: By assumption and definition of F_1, $a \in A \cup \{\mathit{Yes}_{pc}\}$. We have then the following cases:

1. If $a \in A$, then by hypothesis, $a \in B$ and by definition of F_1, $a \in F_1(B, pc')$.
2. If $a = \mathit{Yes}_{pc}$ then we have again two cases:

   $\{pc, pc'\} \subseteq B$: By definition of F_1, $F_1(B, pc') = B \cup \{\mathit{Yes}_p c\}$ and then $a \in F_1(B, pc')$;

   $\{pc, pc'\} \not\subseteq B$: This is not possible, since $\{pc, pc'\} \subseteq A$ and $A \subseteq B$.

$\{pc, pc'\} \not\subseteq A$: By assumption and definition of F_1, $F_1(A, pc') = A \cup \{pc, pc'\}$, thus $a \in A \cup \{pc, pc'\}$. Two cases are possible:

1. If $a \in A$, then by hypothesis, $a \in B$ and by definition of F_1, $a \in F_1(B, pc')$.
2. If $a \in \{pc, pc'\}$ then $a \in B \cup \{pc, pc'\}$, and by definition of F_1, $a \in F_1(B, pc')$.

From all the above cases, we have that $F_1(A, pc') \sqsubseteq F_1(B, pc')$, and hence, that F_1 is monotonic. $\square$

Lemma 11 *F_2 is monotonic.*

Proof. Remember that $(m, pc_m, A) \sqsubseteq_{\mathcal{L}} (n, pc_n, B)$ if and only if $m = n$, $pc_m = pc_n$ and $A \subseteq B$, where $A, B \in \wp(\mathit{ProgCounter} \uplus \mathbb{Y})$. F_2 is the function $F_2 : \wp(\mathit{ProgCounter} \uplus \mathbb{Y}) \times \mathit{ProgCounter} \rightarrow \wp(\mathit{ProgCounter} \uplus \mathbb{Y})$ defined in Eq. (9). Since $m = n$ and $pc_m = pc_n$, we will prove that if $(A, pc') \sqsubseteq (B, pc')$ then $F_2(A, pc') \sqsubseteq F_2(B, pc')$, or what is the same, we will prove that $a \in F_2(A, pc')$ implies $a \in F_2(B, pc')$, assuming $(A, pc') \sqsubseteq (B, pc')$. We make a case-analysis:

$\{pc, pc'\} \subseteq A$: By assumption $\{pc, pc'\} \subseteq B$ and definition of F_2, $F_2(A, pc') = A \setminus \mathbb{Y}_{<pc'} \subseteq B \setminus \mathbb{Y}_{<pc'} = F_2(B, pc')$.

$\{pc, pc'\} \not\subseteq A$: By definition of F_2, $F_2(A, pc') = (A \setminus \mathbb{Y}) \cup \{pc, pc'\}$, so we have the following two cases:

1. If $a \in A \setminus \mathbb{Y}$, then by hypothesis, $a \in B \setminus \mathbb{Y}$ and since $\setminus \mathbb{Y}_{<pc'} \subseteq \mathbb{Y}$ we have that $a \in B \setminus \mathbb{Y}_{<pc'}$. Thus, $a \in F_2(B, pc')$.
2. If $a \in \{pc, pc'\}$ then $a \in (B \setminus \mathbb{Y}) \cup \{pc, pc'\}$, and by definition of F_2, $a \in F_2(B, pc')$.

From all the above cases, we have that $F_2(A, pc') \sqsubseteq F_2(B, pc')$, and hence, that F_2 is monotonic. $\square$

Lemma 12 *F_3 is monotonic.*

Proof. The proof is similar as the proof of Lemma 11, replacing F_3 by F_2 and pc' by $pc+1$. □

Lemma 13 *F is monotonic.*

Proof. Trivial. □

A.2 *Loop*

Lemma 14 (Soundness of *Loop*) *If $Loop_{m,pc_0}$ then $\mathcal{G}_m \models SynC(m, pc_0)$.*

Proof. We need to prove that it exists pc_1 such that $pc_1 \in Reach^+(m, pc_0)$ and $pc_0 \in Reach^+(m, pc_1)$. By definition, $Loop_{m,pc_0}$ iff $Yes \in Loop(m, pc_0)$. We will prove by cases, considering all the cases which could introduce Yes into $Loop(m, pc_0)$.

1. If Yes was introduced or propagated by an application of rule (1), let $pc_0 = pc'$ and $pc_1 = pc$; we show then that $pc \in Reach^+(m, pc')$ (that $pc' \in Reach^+(m, pc)$ is trivial). By hypothesis, $Yes \in Loop(m, pc')$, propagated from $InstrAt(m, pc) = \texttt{goto}\ pc'$ ($pc' \leq pc$). By Lemma 26, Yes_{pc} must belong to $Loop(m, pc')$. Now, if $Yes_{pc} \in Loop(m, pc')$, then by def. of F_1, $\{pc, pc'\} \subseteq Loop(m, pc)$ and $\{pc, pc'\} \subseteq Loop(m, pc')$. By Corollary 18, $pc \in Reach^+(m, pc')$.

2. If Yes was introduced by an application of rule (2), let $pc_0 = pc'$ and $pc_1 = pc$; we show then that $pc \in Reach^+(m, pc')$ (that $pc' \in Reach^+(m, pc)$ is trivial). By hypothesis, $Yes \in Loop(m, pc')$, propagated from $InstrAt(m, pc) = \texttt{goto}\ pc'$ ($pc' > pc$). By Lemma 20, we have that Yes must be equal to $Yes_{pc''}$, $pc'' > pc$ (and $InstrAt(m, pc'') = \texttt{goto}\ pc'''$, with $pc''' < pc''$). By assumption, $Yes_{pc''} \in Loop(m, pc)$, which together with the definition of F_2 implies that $\{pc, pc'\} \subseteq Loop(m, pc)$. By Corollary 18, $pc \in Reach^+(m, pc')$.

3. If Yes was introduced by an application of rule (3), let $pc_1 = pc$ and we have two cases, $pc_0 = pc'$ or $pc_0 = pc + 1$.

   (a) If $pc_0 = pc'$, then the proof is as in the case 1 above;

   (b) If $pc_0 = pc + 1$, we show then that $pc \in Reach^+(m, pc + 1)$ (that $pc + 1 \in Reach^+(m, pc)$ is trivial). By hypothesis, $Yes \in Loop(m, pc + 1)$, propagated from $InstrAt(m, pc) = \texttt{if}\ c\ \texttt{goto}\ pc'$ ($pc' < pc$). By Lemma 20, we have that Yes must be equal to $Yes_{pc''}$, $pc'' > pc$ (and $InstrAt(m, pc'') = \texttt{goto}\ pc'''$, with $pc''' < pc''$). By assumption, $Yes_{pc''} \in Loop(m, pc)$, which together with the definition of F_3 implies that $\{pc, pc + 1\} \subseteq Loop(m, pc)$. By Corollary 19, $pc \in Reach^+(m, pc + 1)$.

4. If Yes was introduced by an application of rule (4) we consider two cases like in the previous item and the proof follows like for the case (2) and the second part of case (3) above.

5. Yes cannot be propagated by rule (6), since we are considering an intra-procedural analysis.

6. The other cases, i.e. if Yes has been propagated by rules (5) and (7) can be reduced to one of the previous cases. □

Lemma 15 (Completeness of *Loop***)** *If $\mathcal{G}_m \models SynC(m, pc)$ then $Loop_{m,pc}$.*

Proof. By hypothesis $\exists pc'' \cdot ((m, pc'') \in Reach^+(m, pc) \wedge (m, pc) \in Reach^+(m, pc''))$. We will prove the lemma by a case-analysis considering $InstrAt(m, pc)$.

1. If $InstrAt(m, pc) = \texttt{goto}\ pc'$ $(pc' < pc)$, first we prove that $(m, pc) \in Reach^+(m, pc')$. By definition of $Reach^+$, any $(m, pc'') \in Reach^+(m, pc)$ must also satisfy $(m, pc'') \in Reach^+(m, pc')$, and together with the hypothesis $(m, pc) \in Reach^+(m, pc'')$, it implies $(m, pc) \in Reach^+(m, pc')$. Hence, we take $pc'' = pc'$. By definition of F_1, $\{pc, pc'\} \subseteq Loop(m, pc')$ and since $(m, pc) \in Reach^+(m, pc')$, by Lemma 22, $\{pc, pc'\} \subseteq Loop(m, pc)$; by definition of F_1, $Yes_{pc} \subseteq Loop(m, pc')$ and by Lemma 24, $Yes \subseteq Loop(m, pc)$.

2. If $InstrAt(m, pc) = \texttt{goto}\ pc'$ $(pc' > pc)$, trivially, $(m, pc') \in Reach^+(m, pc)$ and we can prove that $(m, pc) \in Reach^+(m, pc')$ as in the previous case, so let $pc'' = pc'$. Since $pc' > pc$, it must exist $pc'' > pc$ such that $InstrAt(m, pc'') = \texttt{goto}\ pc'''$ (or $InstrAt(m, pc'') = \texttt{if}\ c\ \texttt{goto}\ pc'''$) with $pc''' \leq pc$, in any sequence of pc-numbers $pc_0 = pc, pc1 = pc', pc_2, \ldots, pc_i = pc'', \ldots, pc_n = pc$, such that $pc_{j+1} \in Reach(m, pc_j)$ (for all $0 \leq j < n$). Let $pc_{i+1} = pc'''$ be the last pc-number in the sequence with the property that $pc''' \leq pc$ and s.t. pc''' is the target of a jump up; pc''' exists by definition of $Reach^+$. By definition of F_1 (F_2 and F_3) and by Lemma 22 we have that $\{pc, pc', pc'', pc'''\} \subseteq Loop(m, pc'')$, $\{pc, pc', pc'', pc'''\} \subseteq Loop(m, pc''')$ and $\{pc, pc', pc'', pc'''\} \subseteq Loop(m, pc)$. By F_1 again, we know that $Yes_{pc''} \in Loop(m, pc'')$ and $Yes_{pc''} \in Loop(m, pc''')$ and by Lemma 24 and our choice of pc'' we have that $Yes \in Loop(m, pc)$.

3. If $InstrAt(m, pc) = \texttt{if}\ c\ \texttt{goto}\ pc'$ $(pc' < pc)$, we consider two cases:
   (a) If $(m, pc') \in Reach^+(m, pc)$ and $(m, pc) \in Reach^+(m, pc')$, then the proof is as the case 1 above;
   (b) If $(m, pc + 1) \in Reach^+(m, pc)$ and $(m, pc) \in Reach^+(m, pc + 1)$, then the proof is analogous as for the case 2 above replacing pc' by $pc + 1$.

4. If $InstrAt(m, pc) = \texttt{if}\ c\ \texttt{goto}\ pc'$ $(pc' > pc)$, we consider two cases:
   (a) If $(m, pc') \in Reach^+(m, pc)$ and $(m, pc) \in Reach^+(m, pc')$; idem case 2 above;
   (b) If $(m, pc + 1) \in Reach^+(m, pc)$ and $(m, pc) \in Reach^+(m, pc + 1)$: the proof is as for the case 3b above.

5. The lemma for the case $InstrAt(m, pc) = \texttt{return}$ is trivially satisfied, since there is no (m, pc') such that $(m, pc') \in Reach^+(m, pc)$ (b).

6. If $InstrAt(m, pc) = \texttt{instr}$, for any other instruction different from the previous ones, the proof maybe reduced to one of the previous cases. □

Lemma 16 *$(m, pc) \in Loop$ if and only if it exists pc' such that at least one of the following conditions hold:*

1. *$InstrAt(m, pc) = \texttt{goto}\ pc'$;*

2. *$InstrAt(m, pc') = \texttt{goto}\ pc$;*

3. *$InstrAt(m, pc) = \texttt{if}\ c\ \texttt{goto}\ pc'$;*

4. $InstrAt(m, pc') =$ `if` c `goto` pc;

5. $InstrAt(m, pc - 1) =$ `if` c `goto` pc';

Proof. The result follows from the fact that pc may have been introduced into $Loop$ only through the application of F_1, F_2 or F_3 at instructions corresponding to one of the cases into consideration. □

Lemma 17 *If $pc \in Loop(m, pc'')$ and $InstrAt(m, pc) =$ `if` c `goto` pc' (or $InstrAt(m, pc) =$ `goto` pc'), then the following holds. It exists a sequence of pc-numbers $pc_0, pc_1, \ldots, pc_n$ $(n \geq 1)$ with $pc_0 = pc$ and $pc_n = pc''$, such that for each $0 \leq i \neq j < n$, $pc_i \neq pc_j$ and for each $1 \leq i \leq n$, $pc \in Loop(m, pc_i)$ and $pc_i \in Reach(m, pc_{i-1})$.*

Proof. By hypothesis and Lemma 16 pc was propagated from $InstrAt(m, pc)$ by an application of rules (1)–(7). Notice, however, that the propagation is done through constraints of the form $f(Loop(m, pc_i)) \sqsubseteq Loop(m, pc_{i+1})$, where $(m, pc_{i+1}) \in Reach(m, pc_i)$. Hence, it exists a sequence of pc-numbers $pc_0 = pc, pc_1, \ldots, pc_n = pc''$ $(n \geq 1)$, such that for all $1 \leq i \leq n$, $pc \in Loop(m, pc_i)$ and $pc_i \in Reach(m, pc_{i-1})$. □

Corollary 18 *If $\{pc, pc'\} \subseteq Loop(m, pc)$ and $InstrAt(m, pc) =$ `if` c `goto` pc' (or $InstrAt(m, pc)$ $=$ `goto` pc') then $pc \in Reach^+(m, pc')$.*

Proof. By Lemma 17 we know there is a sequence of program counters $pc_0, pc_1, \ldots, pc_n$ $(n \geq 1)$ with $pc_0 = pc, pc_1 = pc'$ and $pc_n = pc$, such that for each $1 \leq i \leq n$, $pc_i \in Reach(m, pc_{i-1})$. By definition of $Reach^+$, $pc \in Reach^+(m, pc')$. □

Corollary 19 *If $\{pc, pc + 1\} \subseteq Loop(m, pc)$ and $InstrAt(m, pc) =$ `if` c `goto` pc' then $pc \in Reach^+(pc + 1)$.*

Proof. By Lemma 17 we know there is a sequence of program counters $pc_0, pc_1, \ldots, pc_n$ $(n \geq 1)$ with $pc_0 = pc, pc_1 = pc + 1$ and $pc_n = pc$, such that for each $1 \leq i \leq n$, $pc_i \in Reach(m, pc_{i-1})$. By definition of $Reach^+$, $pc \in Reach^+(m, pc + 1)$. □

Lemma 20 *For any program point pc in method m, if it exists pc' such that $Yes_{pc'} \in Loop(m, pc)$, then $pc \leq pc'$ and $InstrAt(m, pc') =$ `goto` pc'' (or $InstrAt(m, pc) =$ `if` c `goto` pc''), with $pc'' < pc'$.*

Proof. First notice that only an application of F_1 may add $Yes_{pc'}$ to $Loop$, which means that $InstrAt(m, pc') =$ `goto` pc'' (or $InstrAt(m, pc) =$ `if` c `goto` pc''), with $pc'' < pc'$. We only need to prove that $pc \leq pc'$. We consider two cases.

1. If $pc = pc''$, the result follows immediately;

2. If $pc \neq pc''$, then $Yes_{pc'} \in Loop(m, pc)$ has not been introduced by an application of F_1 by it has been propagated by one of the other rules. That $pc \leq pc'$ follows from the fact that $Yes_{pc'}$ has been propagated from $InstrAt(m, pc'')$ to $InstrAt(m, pc)$ by a successive application of rules corresponding to one of the following cases:

    (a) Propagated to the next instruction through $Loop(m, pc) \sqsubseteq Loop(m, pc + 1)$. This is done till reaching $InstrAt(m, pc)$ or to a case reducible to cases 2b or 2c;

(b) Propagated "up" through the application of F_1. Clearly $pc \leq pc'$;

(c) Propagated "down" through the application of F_2. By definition, F_2 filters every $Yes_{pc'''}$ with pc''' bigger than the current program point, hence pc'' cannot be propagated beyond pc';

(d) Propagated "down" through the application of F_3. Like in the previous case, F_3 filters every $Yes_{pc'''}$ with pc''' bigger than the current program point.

From all the above cases we have that $pc \leq pc'$. □

Lemma 21 *For any program point pc in method m, if there exists pc' such that* $Yes_{pc'} \in Loop(m, pc)$*, then* $pc' \in Loop(m, pc)$*.*

Proof. Only an application of F_1 may add $Yes_{pc'}$ to $Loop$, and by definition of F_1, in order to add $Yes_{pc'}$ to $Loop(m, pc'')$, pc' must belong to $Loop(m, pc'')$. Since no rule filters pc-numbers, from $Yes_{pc'} \in Loop(m, pc)$ follows that $pc' \in Loop(m, pc)$. □

Lemma 22 *If* $pc' \in Reach^+(pc)$ *then for any* pc''*,* $pc'' \in Loop(m, pc)$ *implies* $pc'' \in Loop(m, pc')$*.*

Proof. The result follows by definition of *Reach* and the fact that no rule filters pc-numbers (only *Yes* are eventually filtered). □

Lemma 23 *If* $Yes_{pc} \in Loop(m, pc')$ *then* $pc' \in Reach^+(pc)$*.*

Proof. By Lemma 20, $InstrAt(m, pc) = \texttt{goto}\ pc''$ (or $InstrAt(m, pc) = \texttt{if}\ c\ \texttt{goto}\ pc''$), with $pc'' < pc$ and since only an application of F_1 may add Yes_{pc} to $Loop$, we have that $Yes_{pc} \in Loop(m, pc'')$. By definition of *Loop* and *Reach* we have that $pc' \in Reach^+(pc'')$ and hence $pc' \in Reach^+(pc)$. □

Lemma 24 *If* $pc \in Reach^+(pc')$ *and* $Yes_{pc} \in Loop(m, pc')$ *then* $Yes \in Loop(m, pc)$*.*

Proof. By hypothesis, it exists a sequence of pc-numbers $pc_0 = pc', pc_1, \ldots, pc_n = pc$ such that $pc_i \in Reach(pc_{i-1})$ for all $1 < i \leq n$. On the other hand, by Lemma 23 $pc' \in Reach^+(pc)$. We consider two cases:

1. For all $1 < i \leq n$, $pc_i < pc$. In this case Yes_{pc} is propagated from $InstrAt(m, pc')$ to $InstrAt(m, pc)$ since none of the rules (1)–(6) filters $Yes_{pc''}$ for $pc'' < pc$. Thus, $Yes_{pc} \in Loop(m, pc)$.

2. It exists at least one i such that $pc_i > pc$. Let j be the biggest of such i among all the pc-numbers of the sequence $pc_0, pc_1, \ldots, pc_n$. This means that $InstrAt(m, pc_j) = \texttt{if}\ c\ \texttt{goto}\ pc_{j+1}$, with $pc_{j+1} < pc_j$. By rule (1) and definition of F_1, $\{pc_j, pc_{j+1}\} \subseteq Loop(m, pc_{j+1})$. Since $pc' \in Reach^+(pc)$, it follows that $pc_j \in Reach^+(pc_{j+1})$, so by Lemma 22, $\{pc_j, pc_{j+1}\} \subseteq Loop(m, pc_j)$ and by definition of F_1, $Yes_{pc_j} \in Loop(m, pc_{j+1})$. Given that no rule filters Yes_{pc_j} (by assumption pc_j is the biggest pc-number in the sequence $pc_0 = pc', pc_1, \ldots, pc_n = pc$), we have that $Yes_{pc_j} \in Loop(m, pc)$. □

Lemma 25 *If* $Yes_{pc} \in Loop(m, pc')$ *then* $pc \in Reach^+(pc')$*.*

Proof. By hypothesis and Lemma 20 we have that $\mathit{InstrAt}(m, pc) = \texttt{goto}\ pc''$ (or $\mathit{InstrAt}(m, pc) = \texttt{if}\ c\ \texttt{goto}\ pc''$) with $pc'' < pc$ and by definition of F_1, $\{pc, pc''\} \subseteq \mathit{Loop}(m, pc)$ and $\{pc, pc''\} \subseteq \mathit{Loop}(m, pc'')$; by Corollary 18, $pc \in \mathit{Reach}^+(pc'')$. By Lemma 23, $pc' \in \mathit{Reach}^+(pc)$ and $pc' \in \mathit{Reach}^+(pc'')$, thus by Lemma 21, $\{pc, pc''\} \subseteq \mathit{Loop}(m, pc')$.

Suppose, by absurd, that $pc \notin \mathit{Reach}^+(pc')$, then it must exist at least one bifurcation (an `if` instruction) in any sequence $pc_0 = pc'', pc_1, \ldots, pc_n = pc'$ ($pc_{i-1} \in \mathit{Reach}^+(pc_i)$, for any $0 < i \leq n$), such that one branch leads to pc and the other to pc'. Let $\mathit{InstrAt}(m, pc''') = \texttt{if}\ c\ \texttt{goto}\ pc^{iv}$ be the last of such bifurcation instructions; we have the following cases:

1. $pc \in \mathit{Reach}^+(pc^{iv})$ (thus, $pc \in \mathit{Reach}^+(pc'''))$ and $pc' \in \mathit{Reach}^+(pc'''+1)$. By definition of F_3, $\mathit{Yes}_{pc} \in \mathit{Reach}^+(pc'''+1)$ iff $\{pc''', pc'''+1\} \subseteq \mathit{Loop}(m, pc''')$ and again by definition of F_3, $\{pc''', pc'''+1\} \subseteq \mathit{Loop}(m, pc'''+1)$; by Lemma 22, $\{pc''', pc'''+1\} \subseteq \mathit{Loop}(m, pc')$. Now, by our assumption $\mathit{InstrAt}(m, pc''')$ is the last bifurcation between pc'' and pc', hence, by definition of Reach^+ and from $(pc'''+1) \in \mathit{Loop}(m, pc')$ and $(pc'''+1) \in \mathit{Loop}(m, pc''')$ we have that $pc''' \in \mathit{Reach}^+(pc')$ which, together with the fact that $pc \in \mathit{Reach}^+(pc''')$, we have that $pc \in \mathit{Reach}^+(pc')$, which contradicts our original assumption.

2. $pc \in \mathit{Reach}^+(pc'''+1)$ (thus, $pc \in \mathit{Reach}^+(pc'''))$ and $pc' \in \mathit{Reach}^+(pc^{iv})$. We have two sub-cases, depending whether $pc^{iv} > pc'''$ or not.

   (a) If $pc^{iv} > pc'''$, by definition of F_2, $\mathit{Yes}_{pc} \in \mathit{Reach}^+(pc^{iv})$ iff $\{pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc''')$ and again by definition of F_2, $\{pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc^{iv})$. Since $pc' \in \mathit{Reach}^+(pc^{iv})$, by Lemma 22 we have that $\{pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc')$. Since $\mathit{InstrAt}(m, pc''')$ is the last bifurcation between pc'' and pc', and from $\{pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc''')$, we have that $pc''' \in \mathit{Reach}^+(pc')$. By assumption, $pc \in \mathit{Reach}^+(pc''')$ and we have then that $pc \in \mathit{Reach}^+(pc')$, which contradicts our original assumption.

   (b) If $pc^{iv} < pc'''$, by definition of F_1, $\mathit{Yes}_{pc} \in \mathit{Reach}^+(pc^{iv})$ iff $\{pc'', pc, pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc^{iv})$. We know that $\mathit{InstrAt}(m, pc''')$ is the last bifurcation between pc'' and pc', and since $pc' \in \mathit{Reach}^+(pc^{iv})$, by Lemma 22 we have that $\{pc'', pc, pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc')$. Since $\mathit{InstrAt}(m, pc''')$ is the last bifurcation between pc'' and pc', and from $\{pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc''')$ and $\{pc''', pc^{iv}\} \subseteq \mathit{Loop}(m, pc')$ we have, by Corollary 18 that $pc''' \in \mathit{Reach}^+(pc')$. Since $pc \in \mathit{Reach}^+(pc''')$ we have then that $pc \in \mathit{Reach}^+(pc')$, contradicting our original assumption.

From all the above cases, we have a contradiction, thus the lemma holds. □

Lemma 26 *Given* $\mathit{InstrAt}(m, pc) = \texttt{goto}\ pc'$ *($pc' < pc$), if $\mathit{Yes} \in \mathit{Loop}(m, pc)$ and $\mathit{Yes} \in \mathit{Loop}(m, pc')$ then $\mathit{Yes}_{pc} \in \mathit{Loop}(m, pc')$.*

Proof. Assume, by absurd, that $\mathit{Yes}_{pc} \notin \mathit{Loop}(m, pc')$, then by definition of F_1, $pc \notin \mathit{Loop}(m, pc)$ but $pc \in \mathit{Loop}(m, pc')$. By the contra-opposite of Lemma 22, $pc \notin \mathit{Reach}^+(pc')$.

On the other hand, by the hypothesis, it exists pc'' such that $\mathit{Yes}_{pc''} \in \mathit{Loop}(m, pc')$ and $\mathit{Yes}_{pc''} \in \mathit{Loop}(m, pc)$ and by Lemma 20, $\mathit{InstrAt}(m, pc'') = \texttt{goto}\ pc'''$ (or $\mathit{InstrAt}(m, pc'') = \texttt{if}\ c\ \texttt{goto}\ pc'''$) with $pc'' > pc$. By Lemma 21, $pc'' \in \mathit{Loop}(m, pc)$ and $pc'' \in \mathit{Loop}(m, pc')$, thus –by Lemma 17– $pc \in \mathit{Reach}^+(pc'')$ and $pc' \in \mathit{Reach}^+(pc'')$. Moreover, by Lemma 25, $pc'' \in \mathit{Reach}^+(pc')$ and $pc'' \in \mathit{Reach}^+(pc)$. Hence $pc \in \mathit{Reach}^+(pc')$ and we have a contradiction. Thus, the lemma holds. □

A.3 *Loop*'

Lemma 27 (Soundness of *Loop*') *If $Loop'_{m,pc}$ then $\mathcal{G} \models SynCReach(m, pc)$.*

Proof. If $Loop'_{m,pc}$ then $Yes \in Loop'(m, 1)$ and it exists a sequence of method names and *pc*-numbers $(m_0, pc_0), \ldots, (m_i, pc_i), \ldots, (m_n, pc_n)$ such that $InstrAt\ (m_i, pc_i) =$ `invokevirtual` m_{i+1} (for $0 \leq i < n$) and $InstrAt\ (m_n, pc_n) =$ `invokevirtual` m. Since only rule (11) introduce *Yes* into *Loop*', then it must exists $0 < k \leq n$, such that $Yes \in Loop(m_k, 1)$, with $InstrAt\ (m_{k-1}, pc_{k-1}) =$ `invokevirtual` m_k. By soundness of *Loop*, $\mathcal{G}_{m_k} \models SynC(m_k, pc_k)$ and since $\mathcal{G} \models (m, pc) \in Reach^+(m_k, pc_k)$, we have that $SynCReach(m, pc)$. □

Lemma 28 (Completeness of *Loop*') *If $\mathcal{G} \models SynCReach(m, pc)$ then $Loop'_{m,pc}$.*

Proof. By definition of $\mathcal{G} \models SynCReach(m, pc)$, we have that it exists m' such that $\mathcal{G}_{m'} \models SynC(m', pc')$ and $\mathcal{G} \models (m, pc) \in Reach^+(m', pc')$. The result follows by completeness of *Loop* (Lemma 15). □

A.4 *Rec*

Lemma 29 (Soundness of *Rec*) *If $Rec_{m,pc}$ then $\mathcal{M}(P) \models MutRecReach(m)$.*

Proof. First notice that $Yes \in Rec(m, pc)$ iff $Yes \in Rec(m, 1)$. We will prove that it exists a method m' such that $MutRec(m') \wedge m \in Reach^+(m')$. We make the following case analysis:

1. If *Yes* was introduced into *Rec* by rule (15), it exists a *pc*-number *pc* in method m such that $InstrAt\ (m, pc) =$ `invokevirtual` m, thus $m \in Reach(m)$ and $MutRec(m)$.

2. If *Yes* was introduced into *Rec* by rule (16), then it exists (m_k, pc_k) such that *InstrAt* $(m_k, pc_k) =$ `invokevirtual` m ($m \neq m_k$). By rules (15) and (16) we know that $Rec(m_k, pc_k) \neq \emptyset$ and in particular that it contains at least one method name m_i by definition of F. Let $\{m_1, \ldots, m_{k-1}\} \subseteq Rec(m_k, pc_k)$. By Lemma 32 we have that for all $1 \leq i \leq k-1$, $m \in Reach^+(m_i)$. We will show now that among all such m_i, there is m_j such that $MutRec(m_j)$. We consider the following cases.

   (a) Suppose that $Yes \in Rec(m, 1)$ but $Yes \notin Rec(m_k, pc_k)$. By definition of F, $m \in Rec(m_k, pc_k)$ and $m_k \in Rec(m, 1)$ and by Lemma 32 we have that $m_k \in Reach^+(m)$ and $m \in Reach^+(m_k)$, thus $MutRec(m)$.

   (b) Assume now that $Yes \in Rec(m, 1)$ and $Yes \in Rec(m_k, pc_k)$. We know that *Yes* was propagated by an application of F and we assume that $m \notin Rec(m_k, pc_k)$ (if $m \in Rec(m_k, pc_k)$, the proof follows as for the case 2a above). We consider again two cases:

      i. For all $m_i \in Rec(m_k, pc_k)$ ($1 \leq i \leq k-1$), $Yes \in Rec(m_i, 1)$. This is only possible if it exists at least one $m_j \in Rec(m_k, pc_k)$ such that one of the following conditions hold. (i) $InstrAt(m_j, pc_j) =$ `invokevirtual` m_j (for some pc_j in method m_j), so $MutRec(m_j)$ and since by Lemma 32, $m_k \in Reach^+(m_j)$, we have $MutRecReach(m)$. (ii) It exists a method $m_l \in Rec(m_k, pc_k)$ such that $InstrAt(m_j, pc_j) =$ `invokevirtual` m_l (for some pc_j in method m_j) and $m_l \in Rec(m_j, pc_j)$. By Lemma 32, $m_j \in Reach^+(m_l)$ and by definition of *Reach*, $m_l \in Reach^+(m_j)$, so $MutRec(m_l)$ and since $m \in Reach^+(m_l)$ we have that $MutRecReach(m)$.

ii. Suppose now that it exists m_j, m_l in $Rec(m_k)$ such that $InstrAt(m_j, pc_j) =$ `invokevirtual` m_l (for some pc_j in method m_j) and $Yes \in Rec(m_l, 1)$ but $Yes \notin Rec(m_j, 1)$. By a similar argument as the case 2a we prove that $MutRec(m_l)$, and from $m \in Reach^+(m_l)$ we have that $MutRecReach(m)$.

Lemma 30 (Completeness of Rec**)** *If* $\mathcal{M}(P) \models MutRecReach(m)$ *then for any pc-number pc in method* m*,* $Rec_{m,pc}$*.*

Proof. By hypothesis, it exists m' such that $MutRec(m') \wedge m \in Reach^+(m')$. We have to prove that $Yes \in Rec_{m,pc}$. We consider two cases:

- If $m = m'$, then there exists pc such that $InstrAt(m, pc) =$ `invokevirtual` m and by rule (15), $Yes \in Rec_{m,1}$, hence $Yes \in Rec_{m,pc}$ (by propagation from $Rec_{m,1}$ to $Rec_{m,pc}$ by rule (18)).

- If $m \neq m'$, by definition of $MutRec$ we know it exists a sequence of methods $m_0, m_1, \ldots, m_i = m', \ldots, m_n = m$, such that $m_0 \in Reach^+(m_i)$, $m_i \in Reach^+(m_0)$ and $m_n \in Reach^+(m_i)$, and by definition of $Reach$, it exists a sequence of pc-numbers $pc_0, \ldots, pc_i, \ldots, pc_{n-1}$ such that $InstrAt(m_i, pc_i) =$ `invokevirtual` m_{i+1} (for $0 \leq i \leq n-1$). From $m_0 \in Reach^+(m_i)$, $m_i \in Reach^+(m_0)$ and by Lemma 31 we have that, for all j with $0 \leq j \leq i$, $\{m_0, \ldots, m_i\} \subseteq Rec(m_j, pc_j)$; by definition of F, for all $0 \leq j \leq i$, $Yes \in Rec_{m_j, pc_j}$. Now, from $m_n \in Reach^+(m_i)$ and the monotonicity of the rules, we have that $Yes \in Rec_{m_n,1}$, i.e. $Yes \in Rec_{m,1}$ and the result follows. □

Lemma 31 *If* $m' \in Reach^+(m)$ *then for all pc-numbers pc on method* m'*,* $m \in Rec(m', pc)$*.*

Proof. By definition of $Reach^+$ we know it exists a sequence of methods $m_0 = m, m_1, \ldots, m_n = m'$ and of pc-numbers $pc_0, \ldots, pc_i, \ldots, pc_{n-1}$ such that $InstrAt\ (m_i, pc_i) =$ `invokevirtual` m_{i+1} (for $0 \leq i \leq n-1$). By definition of rules (15) and (16), for all $0 \leq i < n$ $m_i \in Rec(m_{i+1}, pc)$, for all pc-number on method m_{i+1}. By monotonicity of the rules (15)–(18), we have that for all $0 \leq j \leq n$, $m_0 = m \in Rec(m_j, pc)$ (for for all pc-number on method m_j). Thus, $m \in Rec(m', pc)$ (for for all pc-number on method m'). □

Lemma 32 *If* $m \in Rec(m', pc)$ *for some pc, then* $m' \in Reach^+(m)$*.*

Proof. Suppose, by absurd, that $m' \notin Reach^+(m)$, which means there is no sequence of methods $m_1, \ldots, m_k$ such that $m_1 = m$, $m_k = m'$ and $InstrAt(m_i, pc_i) =$ `invokevirtual` m_{i+1} for all $1 \leq i < k$. But, by definition of $Reach^+$ and rules (15) and (16), m cannot have been added to $Rec(m')$, contradicting the hypothesis. □

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399