Constructing a Java Card converter using a proof assistant[a]

Thomas Genet, Thomas Jensen and François Monin

Projet Lande

IRISA

---

[a]Partly supported by the INRIA research action"S-Java" and by EU IST R&D project "VerifiCard"

Overview

Goal: Formal construction of program transformers aimed at compressing code to fit onto embedded systems.

Case study: Transformation of Java Card byte code according to the CAP format.

Formalisation: Type-structured description of un-optimised and optimised format.

Overview

- Tokenisation and the CAP format.
- Formalism for relating class files and CAP files.
- A reduced PVS model for studying tokenisation.

The CAP format

Optimised representation of Java Card class files:

$$\text{Java package} \xrightarrow{compiler} \text{class files} \xrightarrow{CAP\ converter} \text{CAP file}$$

Tokenisation: Replace Java names (strings) with tokens (integers)

Componentisation: Join class files of a package for more direct addressing. One constant pool, method component, class component *etc* per package.

CAP method tables

Method tables: a fast implementation of dynamic method lookup.

Tables provide pointers to the code of inherited and defined methods of a class.

If a method definition $B.m$ is a **re-definition** of a method $A.m$ from a super-class, $B.m$ takes the place of $A.m$ in the table.

CAP method tables

```
class A {
 foo() { <code1> } ;
 bar() { <code2> } ;
}
```

foo	&code1
bar	&code2

```
class B extends A {
 bar() { <code3> } ;
 baz() { <code4> } ;
}
```

foo	&code1
bar	&code3
baz	&code4

CAP method tables

Each class has two method tables:

- public methods (visible outside defining package)
- package methods (visible only within defining package).

Reduced tables for mixing direct method lookup and dynamic method search:

$\texttt{token} < \texttt{table\_base} \Rightarrow$ “search super class’ method table”

Tokenisation of methods

Replace a method name by a number (its token).

The token corresponds to its **index** in the method table.

A method that redefines another method receives **the same** token as the method it redefines.

Methods in super classes have smaller tokens than methods in their sub-classes.

Correct method tokenisation I

Overriding methods get the same token:

$$\left.\begin{array}{l}
\langle c_name, \textcolor{red}{sig}\rangle \; T \; \langle c_ref, m_tok\rangle \;\wedge \\
\langle c'_name, \textcolor{red}{sig}\rangle \; T \; \langle c'_ref, m'_tok\rangle \;\wedge \\
c'_name \textcolor{red}{<} c_name \;\wedge \\
(package(sig) \Rightarrow same_package(c_name, c'_name))
\end{array}\right\} \Rightarrow \begin{array}{c} m_tok \\ = \\ m'_tok \end{array}$$

Correct method tokenisation II

Numbering respects the class hierarchy:

$$\left.\begin{array}{l} public(sig) \ \wedge \ new\_method(sig, c\_name) \wedge \\ (c\_name, sig) \ T \ \ (c\_ref\,, m\_tok) \end{array}\right\} \Rightarrow$$

$$\forall m'\_tok \in super(c\_ref)\,.\, m\_tok > m'\_tok$$

A reduced model in PVS

Model only concerned with tokenisation of public methods—can ignore the package structure.

No componentisation - Ignore constant pools and the method code.

Spécifications

STree : TYPE = $\{$p : CTree $|$ classupsub(p) and Onlynameclass(p) $\}$

Cond1 : LEMMA FORALL (P: STree[ClassRefName, MethodName],
c1,c2 : ClassRefName, m: MethodName) :
Occursintree(c1,P) and Occursintree(c2,P) and
membermeth_in_class(m,c1,P) and membermeth_in_class(m,c2,P)
and infeq_class(c1,c2,P)

$$\Longrightarrow$$

f_tok(P,c1,m)=f_tok(P,c2,m)

Spécifications

STree : TYPE = $\{$p : CTree $|$ classupsub(p) and Onlynameclass(p) $\}$

Cond2 : LEMMA FORALL (P: STree[ClassRefName, MethodName],
c1,c2 : ClassRefName, m1,m2: MethodName) :
Occursintree(c1,P) and Occursintree(c2,P) and
membermeth_in_class(m1,c1,P) and
membermeth_in_class(m2,c2,P) and infeq_class(c1,c2,P)

$$\Longrightarrow$$

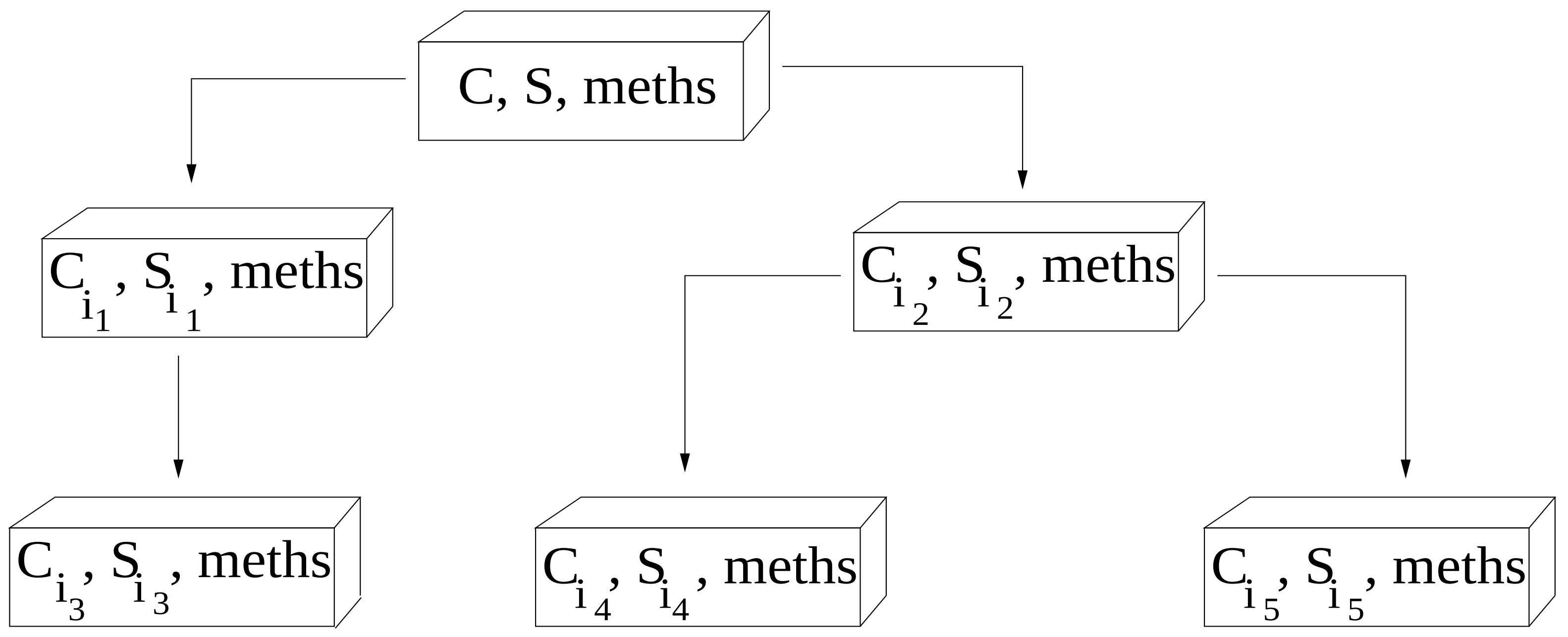
f_tok(P,c2,m2) $\leq$ f_tok(P,c1,m1)

Generic packages, class files and CAP files

Package : tree/forest.

class file : class, superclass, methods.

CAP file : class, superclass, tokens.



Generic class files, CAP files

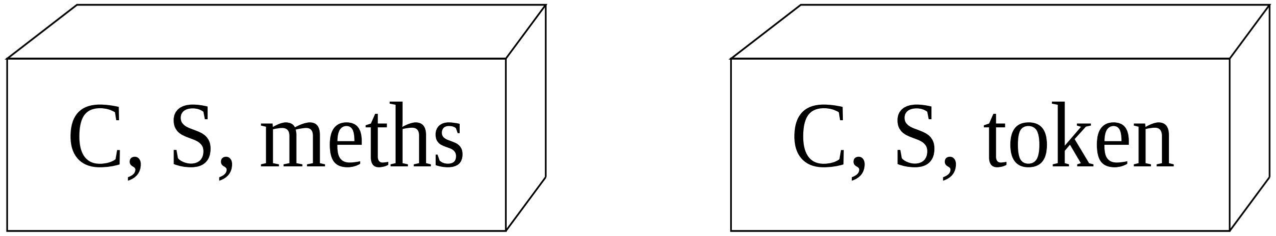
CNode[CR: TYPE, MN: TYPE]: DATATYPE

BEGIN

classNode(className: CR, super: CR, the_elements: list[MN]):

cnode?

END CNode



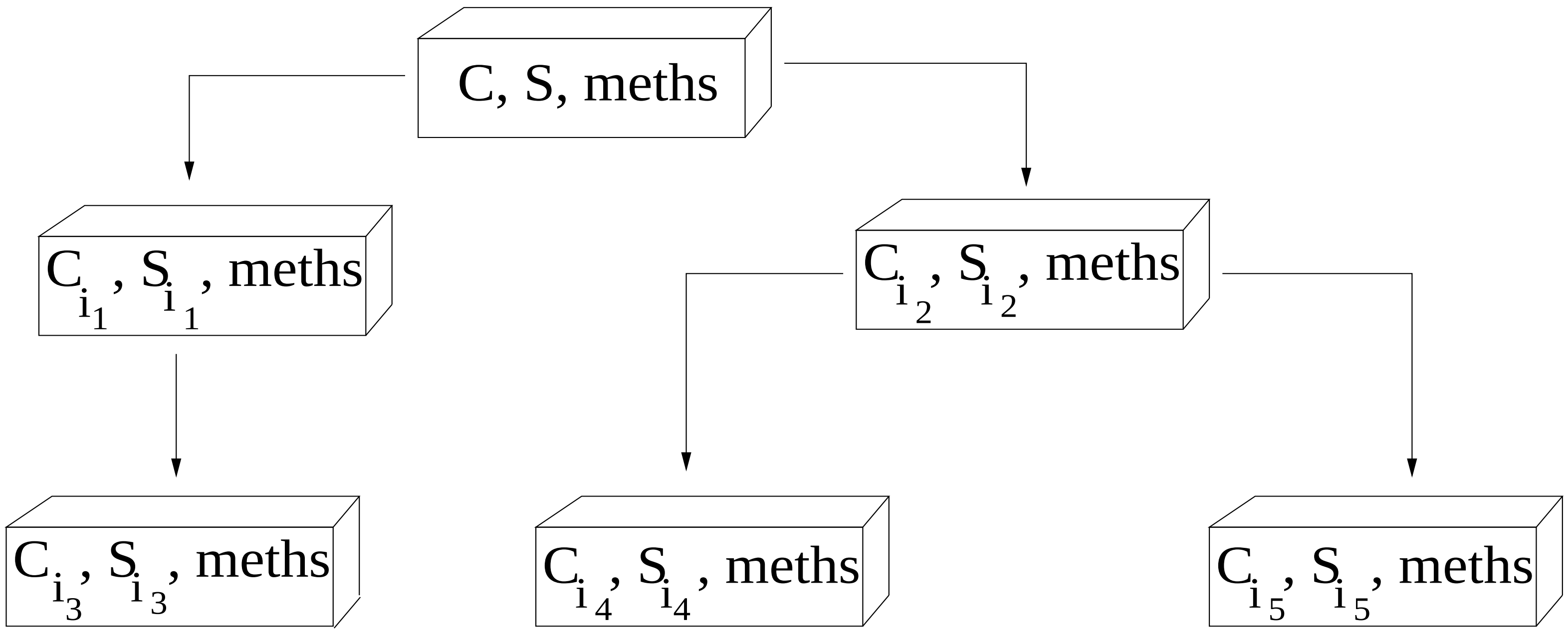
Generic Packages

CTree[CR: TYPE, MN: TYPE] : DATATYPE

BEGIN IMPORTING CNode[CR,MN]

nodeclass(classtree: CNode, subclass: list[CTree]): nodeclass?

END CTree



Specification of superclasses

```
supsub (cl: CR, l: list[CTree]): RECURSIVE bool =
      . . .
MEASURE (lambda (cl: CR, l: list[CTree]) : length(l))


classuplist(l: list[CTree]) : RECURSIVE bool =
  CASES l OF
      null : TRUE,
      cons(e,res) : IF not(supsub(className(classtree(e)),subclass(e)))
                      THEN FALSE ELSE
            IF not(classuplist(subclass(e))) THEN FALSE
                 ELSE classuplist(res) ENDIF ENDIF ENDCASES
MEASURE lengthlistCtree
```

Specification of superclasses

Classupsub(p: CTree) : bool =

Let scl = className(classtree(p)), sub = subclass(p) IN

IF not(supsub(scl, sub)) THEN FALSE ELSE classuplist(sub) ENDIF

Onlynameclass (p: CTree): bool = . . .

STree : TYPE = $\{$ p : CTree $|$ Classupsub(p) and Onlynameclass(p) $\}$

The Converter

Two main functions:

- The convert function
- The tokenize function:

What does the tokenize function do?

MethodNames: (a$_{i_1}$; a$_{i_2}$; . . . ; a$_{i_m}$)

$*$

Functok: ((b$_{j_1}$, s$_{j_1}$); (b$_{j_2}$, s$_{j_2}$); . . . ; (b$_{j_n}$, s$_{j_n}$))

$*$

Tokens: (t$_{k_1}$; t$_{k_2}$; . . . ; t$_{k_p}$)

$*$

Token: t

$\longrightarrow$ Functok $*$ Tokens $*$ Token

The tokenize function

```
        tokenize
(mlist: list[MethodName], conv: FuncTok, tl: list[Token], i: nat) :
RECURSIVE [FuncTok,list[Token],nat] =
  CASES mlist OF
      null: (conv,tl,i),
      cons(methname,rem) : LET t=get_assoc(methname,conv) IN
  CASES t OF
none : tokenize(rem, cons((methname,i), conv), cons(i,tl), i+1),
res(j) : tokenize(rem, conv, cons(j,tl), i)
    ENDCASES
ENDCASES
MEASURE length(mlist)
```

The convert function

convertlist

(testlist: bool, cflist: list[CTree[ClassRefName, MethodName]], conv:

Converter, i:nat):

RECURSIVE [list[CTree[ClassRefName, Token]], Converter, nat] = . . .

It must be total.

It cannot be mutual recursively defined.

converter(p: CTree[ClassRefName, MethodName]):

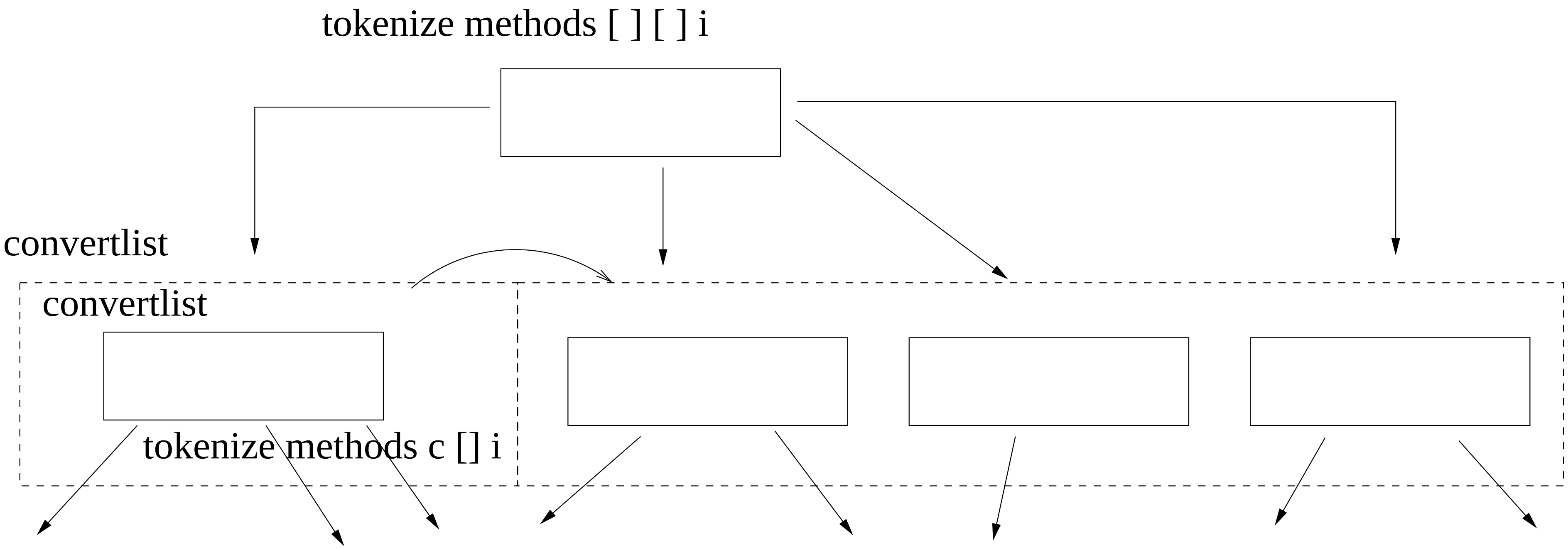
[list[CTree[ClassRefName,Token]], Converter, nat]

=

convertlist(true, cons(p, null), null, 1)

The convert function

(testlist: bool, cflist: list[CTree], conv: Converter, i:nat)



Correctness properties

Type Correctness Conditions (TCCs).

Type properties.

Auxiliary functions properties.

Invariant properties.