Modular proof principles for parameterised concretizations

David Pichardie

IRISA / ENS Cachan (Bretagne)
IRISA, Campus de Beaulieu
F-35042 Rennes, France
david.pichardie@irisa.fr

Abstract. Abstract interpretation is a particularly well-suited methodology to build modular correctness proof of static analysers. Proof modularity becomes essential when correctness proof is machine checked for realistic languages To deal with complex concrete and abstract domains, the notion of parameterised concretization has been proposed to allow a structural decomposition of the abstract domain and its concretization. In this paper we develop proof principles for such concretizations, based on the theoretical notion of concretization functor, with the aim of obtaining modular correctness proofs. Our technique has been tested on a machine-checked correctness proof of a static analysis for a Java-like bytecode language.

1 Introduction

Machine-assisted deductive methods improve the reliability of analysers, by providing machine-checked correctness proofs from which implementations of analysers are automatically *extracted*. The feasibility of the approach was demonstrated in a previous paper [3], but the human cost of such a work remains a major drawback to develop a large number of such *certified* static analysers. In [3], a first basis of a generic framework for proving and extracting static analysers in the Coq [5] proof assistant was proposed but this reusable part was mainly dedicated to the specification of the analysis and the extraction of the analyser. The correctness proof of the abstract semantic with respect to the standard semantics was done in an *ad hoc* fashion due to a lack of methodology. This paper aims at improving this point by proposing proof techniques that allow to modularise such proofs. The technical concept underlying these techniques is that of *parameterised concretisation functions*.

Abstract interpretation proposes a rich mathematical framework for conducting such correctness proofs of static analysers. It is particularly well-suited to propose modular and generic construction usable for several analyses and programming languages. It is then a very promising tool when dealing with machine checked proof. In this context proof are done *in-extenso* with a high level of detail. The global architecture of the proof becomes then a critical point, specially when dealing with static analysis of "real" languages.

A simple example of modular technique is the abstraction product. To abstract a concrete domain of the form $\mathcal{P}(C \times D)$ a simple modular approach is to split the proof into two distinct parts : an abstract domain $C^\sharp$ to abstract $\mathcal{P}(C)$ (using a monotone concretization function $\gamma^C \in \mathcal{P}(C) \to C^\sharp$) and an abstract domain $D^\sharp$ to abstract $\mathcal{P}(D)$ (using $\gamma^D \in \mathcal{P}(D) \to D^\sharp$). Each abstraction can then be developed and proved correct forgetting the other. Global abstraction is then done on the product domain $C^\sharp \times D^\sharp$ with a concretization γ defined by

$$\forall (c^\sharp, d^\sharp) \in C^\sharp \times D^\sharp,\ \gamma\left(c^\sharp, d^\sharp\right) = \left\{ (c,d) \;\middle|\; \begin{array}{l} c \in \gamma^C(c^\sharp) \\ d \in \gamma^D(d^\sharp) \end{array} \right\}$$

This technique is particularly tempting for a real language like Java bytecode whose memory space looks like Heap$\times$Static Heap$\times$Operand Stack$\times$Local Variable. This technique allows hence to split the proof effort into four independent parts. Unfortunately this modular technique restrict enormously the power of the abstraction usable because it necessarily forgets any relation on $C \times D$. On the other side, full relational abstractions compute properties on $C \times D$ but are difficult to modularize. In this paper we study a restricted class of relational abstraction, called *parameterised* where a concretization function can be parameterised by a concrete element. For example, for the analysis of heap structure, the concretization for reference sometimes only makes sense in the context of a concrete heap. At the global abstraction level, the concretization is then of the form

$$\forall (c^\sharp, d^\sharp) \in C^\sharp \times D^\sharp,\ \gamma\left(c^\sharp, d^\sharp\right) = \left\{ (c,d) \;\middle|\; \begin{array}{l} c \in \gamma^C(c^\sharp) \\ d \in \gamma^D_c(d^\sharp) \end{array} \right\}$$

As we will see in Section 4, this dependence of γ^D with respect to C is one obstacle for proof modularity. The main contribution of this paper is to propose a modular proof technique compatible with parameterised concretization. This proof technique is based on a natural notion of concretization functor. The technique require some restriction on the used abstraction but we have nevertheless been able to experiment it on a realistic representation of bytecode language with non-trivial abstractions. The whole proof of a generic static analysis has been machine-checked using this technique. The Coq source of development are available on-line at http://www.irisa.fr/lande/pichardie/CarmelCoq.

Plan of the paper. Our machine-checked proof concerns a language similar to the Java byte code, named Carmel, presented in Section 2. In Section 3, we present classic modular constructions which appear to be difficult to use with concretization functions (presented in Section 4). We then propose a notion of concretization functor in Section 5 and shows it modularity capabilities. The machine-checked proof is briefly described in Section 6. Section 7 presents the relative work and Section 8 concludes.

Notations and prerequisites. We write :: for the list concatenation symbol, A^+ represents the set of non empty sequences of elements in a set A, $\to_m$ denotes the monotone functions constructor and $\rightharpoonup$ the partial functions constructor. The pointed notation on order symbol ($\dot{\sqsubseteq}$) represents the associated point-wise extension of the order ($f_1 \dot{\sqsubseteq} f_2 \stackrel{\text{def}}{\iff} \forall x, f_1(x) \sqsubseteq f_2(x)$). We assume basic knowledge of abstract interpretation [8] concepts such as concretization function and partial trace semantics.

2 Target case study

The notion of parameterised concretisation functions is not linked to a particular programming language or abstraction, but we have chosen to present our results in the concrete setting of a representative subset of the Carmel language [10,3]. The language is a bytecode for a stack-oriented machine, much like the Java Card bytecode. We concentrate here on the intraprocedural fragment with instructions about stack operations, numeric operations, conditionals, object creation and modification. We leave out method calls which are not needed to explain our results and which would only complicate the presentation. Thus, the role of objects are reduced to dynamically allocated records. Nevertheless, the semantic domain includes a heap and an environment and is sufficiently complex to test our proof modularization technique.

$$
\begin{array}{rlrl}
\text{Val} &= \mathbb{N} + \text{Reference} + \{\text{null}\} & & \\
\text{LocalVar} &= \text{Var} \rightharpoonup \text{Val} & \text{Stack} &= \text{Val}^* \\
\text{Object} &= \text{ClassName} \times (\text{FieldName} \rightharpoonup \text{Val}) & \text{Heap} &= \text{Reference} \rightharpoonup \text{Object} \\
\text{State} &= \text{ProgPoint} \times \text{Heap} \times \text{LocalVar} \times \text{Stack} & &
\end{array}
$$

Fig. 1. Carmel semantic domains

The language is given a small-step operational semantics manipulating states of the form $\langle\!\langle pc, h, l, s \rangle\!\rangle$, where pc is a program point, h a heap of objects, l a set of local variables, and s a local operand stack (see [14] or [4] for details). Formal definitions of the semantic domains are given in Figure 1 and the different semantic rules are presented in Figure 2. We write $s_1 \rightarrow_i s_2$ if s_2 is the new state resulting from the execution of instruction i in state s_1. The values we manipulate are either integers or memory references. We let n ranges over integers and loc over references. The instruction `numop` is parameterised by an operator name op (addition, multiplication, ...) whose semantics is given by $[\![op]\!]$. The value stored in the local variable x is represented by $l[x]$ (see instruction `load`). $l[x \mapsto v]$ assigns the variable x to the value v and leaves the others values in l unchanged (similar notations are used for heaps and objects). Two rules define the `if` instruction behavior according to the first value of the current operand stack. The last three instructions deal with object manipulation. The function newObject computes, for a class name cl and a heap h, a new memory reference loc where a new object $\text{def}(cl)$ of class cl will be stored. The notation $o.f$ represents the access to a field f in the class instance o (f should be a declared field of the class of o, see condition $f \in \text{definedFields}(\text{class}(o))$).

$\langle\!\langle pc, h, l, s\rangle\!\rangle \rightarrow_{\texttt{nop}} \langle\!\langle pc+1, h, l, s\rangle\!\rangle$	$\langle\!\langle pc, h, l, v :: s\rangle\!\rangle \rightarrow_{\texttt{pop}} \langle\!\langle pc+1, h, l, s\rangle\!\rangle$
$\langle\!\langle pc, h, l, s\rangle\!\rangle \rightarrow_{\texttt{push}\ n} \langle\!\langle pc+1, h, l, n :: s\rangle\!\rangle$	$\langle\!\langle pc, h, l, s\rangle\!\rangle \rightarrow_{\texttt{goto}\ pc'} \langle\!\langle pc', h, l, s\rangle\!\rangle$
$\langle\!\langle pc, h, l, n_1 :: n_2 :: s\rangle\!\rangle \rightarrow_{\texttt{numop}\ op} \langle\!\langle pc+1, h, l, [\![op]\!](n_1, n_2) :: s\rangle\!\rangle$	
$\langle\!\langle pc, h, l, s\rangle\!\rangle \rightarrow_{\texttt{load}\ x} \langle\!\langle pc+1, h, l, l[x] :: s\rangle\!\rangle$	
$\langle\!\langle pc, h, l, v :: s\rangle\!\rangle \rightarrow_{\texttt{store}\ x} \langle\!\langle pc+1, h, l[x \mapsto v], s\rangle\!\rangle$	
$\langle\!\langle pc, h, l, n :: s\rangle\!\rangle \rightarrow_{\texttt{if}\ pc'} \langle\!\langle pc+1, h, l, s\rangle\!\rangle$ when $n = 0$	$\langle\!\langle pc, h, l, n :: s\rangle\!\rangle \rightarrow_{\texttt{if}\ pc'} \langle\!\langle pc', h, l, s\rangle\!\rangle$ when $n \neq 0$
$\langle\!\langle pc, h, l, s\rangle\!\rangle \rightarrow_{\texttt{new}\ cl} \langle\!\langle pc+1, h[loc \mapsto \mathrm{def}(cl)], l, loc :: s\rangle\!\rangle$ when $\exists c \in \mathrm{classes}(P)$ with $\mathrm{ClassName}(c) = cl$ and $loc = \mathrm{newObject}(cl, h)$	
$\langle\!\langle pc, h, l, v :: loc :: s\rangle\!\rangle \rightarrow_{\texttt{putfield}\ f} \langle\!\langle pc+1, h[loc \mapsto o'], l, s\rangle\!\rangle$ when $h(loc) = o,\ f \in \mathrm{definedFields}(\mathrm{class}(o))$ and $o' = o[f \mapsto v]$	
$\langle\!\langle pc, h, l, loc :: s\rangle\!\rangle \rightarrow_{\texttt{getfield}\ f} \langle\!\langle pc+1, h, l, o.f :: s\rangle\!\rangle$ when $h(loc) = o$ and $f \in \mathrm{definedFields}(\mathrm{class}(o))$	

Fig. 2. Operational semantic rules of Carmel

The *partial trace semantics* $[\![P]\!]$ of a Carmel program P is defined as the set of reachable partial traces:

$$[\![P]\!] = \left\{ s_0 s_1 \cdots s_n \in \mathrm{State}^+ \;\middle|\; \begin{array}{l} s_0 \in \mathcal{S}_{init} \wedge \\ \forall k < n,\ \exists i,\ s_k \rightarrow_i s_{k+1} \end{array} \right\} \in \mathcal{P}(\mathrm{State}^+)$$

where $\mathcal{S}_{init}$ is the set of initial states.

The goal of the analysis is to compute an approximation of $[\![P]\!]$ for any given program P. The approximation lives in an abstract domain $\mathcal{D}^\sharp$ with a poset structure $(\mathcal{D}^\sharp, \sqsubseteq)$. The correctness of the approximation is specified by a monotone concretization function γ belonging to $(\mathcal{D}^\sharp, \sqsubseteq) \rightarrow_m (\mathcal{P}(\mathcal{D}), \subseteq)$. All these elements form what we called a *connection* (in reference to Galois connections whose abstraction function is nevertheless not explicitly used in this paper).

For simplicity, example taken in Section 3, 4 and 5 will not be directly related to Carmel. They will nevertheless be inspired by the analysis effectively proved in Coq and presented in Section 6.

3 Modular construction of connection

The abstract interpretation theory explains how compose connections to build new connections from old. A classic example of such a construction is the abstraction of variable environments (partial map from variable name to value) which can be constructed for any abstraction of values.

Definition 1. ***(Generic environment connection)*** *We define the term* generic environment connection *to mean a functional which maps a connection* $(\mathcal{P}(\mathit{Val}), \subseteq) \xleftarrow{\gamma^{\mathit{Val}}} \left(\mathit{Val}^\sharp, \sqsubseteq_{\mathit{Val}^\sharp}\right)$ *to a 5-upplet* $\left(\mathit{Env}^\sharp, \sqsubseteq_{\mathit{Env}^\sharp}, \gamma_{\mathit{Env}}, \mathrm{get}^\sharp, \mathrm{subst}^\sharp\right)$ *with*

- $\left(\mathit{Env}^\sharp, \sqsubseteq_{\mathit{Env}^\sharp}\right)$ *a partially ordered set*
- $\gamma \in \left(\mathit{Env}^\sharp, \sqsubseteq_{\mathit{Env}^\sharp}\right) \to_m (\mathcal{P}(\mathit{Var} \to \mathit{Val}), \subseteq)$ *a monotone concretization function between* $\mathit{Env}^\sharp$ *and the set of environment*
- $\mathrm{get}^\sharp \in \mathit{Env}^\sharp \times \mathit{Var} \to \mathit{Val}^\sharp$ *a correct approximation of the function giving the value attached with each variable*

$$\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var}, \quad \left\{ \rho(x) \;\middle|\; \rho \in \gamma(\rho^\sharp) \right\} \subseteq \gamma^{\mathit{Val}}(\mathrm{get}^\sharp(\rho^\sharp, x))$$

- $\mathrm{subst}^\sharp \in \mathit{Env}^\sharp \times \mathit{Var} \times \mathit{Val}^\sharp \to \mathit{Env}^\sharp$ *a correct approximation of the function which substitute a value with an other one in a variable*

$$\begin{aligned}&\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\ \forall v^\sharp \in \mathit{Val}^\sharp,\\ &\qquad \left\{ \rho[x \mapsto v] \;\middle|\; \begin{array}{l} \rho \in \gamma(\rho^\sharp) \\ v \in \gamma^{\mathit{Val}}(v^\sharp) \end{array} \right\} \subseteq \gamma(\mathrm{subst}^\sharp(\rho^\sharp, x, v^\sharp))\end{aligned}$$

Hence a generic environment connection constructs an abstract domain, a concretization function and two correct approximations of the primitive function for manipulating environments, given a connection for abstracting values.

An example of such connection constructor is given by the classical non-relational abstraction.

Lemma 1. *The functional which associates with all connection* $(\mathcal{P}(\mathit{Val}), \subseteq) \xleftarrow{\gamma^{\mathit{Val}}} \left(\mathit{Val}^\sharp, \sqsubseteq_{\mathit{Val}^\sharp}\right)$ *the 5-upplet* $\left(\mathit{Env}^\sharp, \sqsubseteq_{\mathit{Env}^\sharp}, \gamma_{\mathit{Env}}, \mathrm{get}^\sharp, \mathrm{subst}^\sharp\right)$ *with*

- $\mathit{Env}^\sharp = \mathit{Var} \to \mathit{Val}^\sharp$
- $\sqsubseteq_{\mathit{Env}^\sharp} = \dot{\sqsubseteq}_{\mathit{Val}}$
- $\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \gamma_{\mathit{Env}}(\rho^\sharp) = \left\{ \rho \;\middle|\; \forall x \in \mathit{Var},\ \rho(x) \in \gamma^{\mathit{Val}}(\rho^\sharp(x)) \right\}$
- $\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\ \mathrm{get}^\sharp(\rho^\sharp, x) = \rho^\sharp(x)$
- $\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\ \forall v^\sharp \in \mathit{Val}^\sharp,\ \mathrm{subst}^\sharp(\rho^\sharp, x, v^\sharp) = \rho^\sharp[x \mapsto v^\sharp]$

is a correct generic environment.

This lemma expresses that the non-relational abstraction of environments can be constructed for any abstraction of values. Hence, several value abstractions can be used without having to redo any proof about abstract environments. Such a generic construct has an additional advantage when working with a proof assistant : during its construction, the value abstraction is opaque and hence the proof is simpler, only focusing on environment manipulations. It is thus particularly convenient to use such generic constructors for a machine-checked proof but they are unfortunately difficult to use for more sophisticated value abtractions. In particular, analyses of the *heap* structure (or the memory) of dynamically allocated data structures (references, cells, objects, ...) can required other form of connections.

Example 1. If all values in the language are reference on dynamically allocated object in a heap, an abstraction of these references by the set of class name of the associated object only makes sense in the context of a concrete heap.

$$\forall s \in \mathcal{P}(\text{Class}),\quad \gamma(s) = \left\{ (h, \text{loc}) \;\middle|\; \text{loc} \in \text{dom}(h) \;\wedge\; \text{class}(h(\text{loc})) \in s \right\} \subseteq \text{Heap} \times \text{Val}$$

with Heap and Object defined as for Carmel semantic domains.

This kind of concretization is generally written in a nicer, parameterised form

$$\forall h \in \text{Heap},\ \forall s \in \mathcal{P}(\text{Class}),\quad \gamma_h(s) = \left\{ \text{loc} \;\middle|\; \text{loc} \in \text{dom}(h) \;\wedge\; \text{class}(h(\text{loc})) \in s \right\} \subseteq \text{Val}$$

We will now formally define this kind of concretization and show how we can use them during correctness proofs.

4 Parameterised concretization

The concretization function we study here depends on a context. Each abstract value is concretized into a relation between a concrete value and a context element, where the context element is necessary to give a non-trivial concretisation of the abstract element. We are hence interested in connection of the following form

$$(\mathcal{P}(C \times D), \subseteq) \xleftarrow{\gamma} (D^\sharp, \sqsubseteq)$$

with C the *context domain*. Some examples:

Example 2. The same kind of concretization as in example 1 can be used to abstract references by the super-class of all objects they refer (this is the abstraction taken in the Java bytecode verifier).

$$\forall \tau \in \text{Class},\quad \gamma(\tau) = \left\{ (h, \text{loc}) \;\middle|\; \text{loc} \in \text{dom}(h) \;\wedge\; \text{class}(h(\text{loc})) \prec_P \tau \right\} \subseteq \text{Heap} \times \text{Val}$$

where $\prec_P$ is the subtyping relation associated with the class hierarchy of program P.

Example 3. A more precise abstraction than abstraction by set of class names can be obtained by abstracting with set of creation point. The formal definition of the concretization function is then relative to a partial execution trace.

A partial trace is a non-empty sequence $< pc_0, m_0 >:: \cdots ::< pc_n, m_n >$ of states, each state containing a program point pc_i (taken in a set ProgPoint) and a memory m_i. If the instruction found at a program point pc is an object creation with class cl (event noted $\text{instr}(pc) = \texttt{new}\ cl$), a new adress $\text{newObject}(cl, m)$ is allocated in the memory m to stock an object of class cl inside.

The associated concretization is

$$\forall s \in \mathcal{P}(\text{ProgPoint}),\quad \gamma(s) = \left\{ (< pc_0, m_0 >:: \cdots ::< pc_n, m_n >, \text{loc}) \;\middle|\; \begin{array}{l} \exists k \in \{0, \ldots, n\}, \\ \quad pc_k \in s \\ \quad \text{instr}(pc_k) = \texttt{new}\ cl \\ \quad \text{newObject}(cl, m_k) = \text{loc} \end{array} \right\}$$

End of examples.

This kind of concretization can be represented under a equivalent **parameterised** form. We will note γ^{param} the function of $C \to D^\sharp \to \mathcal{P}(D)$ defined by

$$\forall c \in C,\ \forall d^\sharp \in D^\sharp,\ \gamma_c^{\text{param}}(d^\sharp) = \{d \mid (c,d) \in \gamma(d^\sharp)\}$$

Most of the time, we will omit the $\cdot^{\text{param}}$ notation because the context will allow us to do it without ambiguity.

4.1 Using generic connections with parameterised concretization

When fixing an element $c \in C$ in the context, we can treat γ_c as a concretization in $D^\sharp \to_m \mathcal{P}(D)$, forgetting the relational view. We are then back to the application framework of the modular construction exposed in the previous section: a parameterised abstraction $\mathcal{P}(\text{Val}) \xleftarrow{\gamma_c^{\text{Val}}} \left(\text{Val}^\sharp, \sqsubseteq_{\text{Val}^\sharp}\right)$ (with c a fixed element in C) can be used to instantiate any generic environment connection. We obtain a collection of 5-upplet $\left(\text{Env}^\sharp, \sqsubseteq_{\text{Env}^\sharp}, \gamma_c^{\text{Env}}, \text{get}^\sharp, \text{subst}^\sharp\right)_{c\in C}$ with $\text{get}^\sharp$ for example verifying

$$\forall c \in C,\ \forall \rho^\sharp \in \text{Env}^\sharp,\ \forall x \in \text{Var},\quad \left\{\rho(x) \;\middle|\; \rho \in \gamma_c^{\text{Env}}(\rho^\sharp)\right\} \;\subseteq\; \gamma_c^{\text{Val}}(\text{get}^\sharp(\rho^\sharp, x))$$

A generic environment connection is then able to use a parameterised value concretization to produce a parameterised environment concretization with its correct basic operators. Nevertheless, note that the correctness property assured by these operators are relative to the same context c. As we will see now this will be a major limitation when proving correctness of abstract transfer functions.

4.2 Proving correctness of abstract transfer functions

The difficulties with parameterised concretisations become apparent when we consider proving the correctness of transfer functions (the abstract interpretation of each byte code). For example, in a language with variables and dynamic allocations the memory state is of the form $\text{Mem} \stackrel{\text{def}}{=} \text{Heap} \times \text{Env}$ with $\text{Heap} \stackrel{\text{def}}{=} \text{Val} \nrightarrow \text{Object}$, $\text{Env} \stackrel{\text{def}}{=} \text{Var} \nrightarrow \text{Val}$ and Val the domain value, reduced here at addresses in the heap.

Because the memory is split into two different structures, it is natural to abstract it with two distinct abstract elements. Given a heap connection $(\mathcal{P}(\text{Heap}), \subseteq) \xleftarrow{\gamma^{\text{Heap}}} (\text{Heap}^\sharp, \sqsubseteq_{\text{Heap}})$ and for the variable environments, let suppose the value abstraction has required a heap parametrization (as in example 1): the abstraction is hence of the form

$$\left((\mathcal{P}(\text{Env}), \subseteq) \xleftarrow{\gamma_h^{\text{Env}}} (\text{Env}^\sharp, \sqsubseteq_{\text{Env}})\right)_{h\in\text{Heap}}$$

At the memory level, the concretization of a couple $(h^\sharp, \rho^\sharp)$ of abstract elements will be

$$\gamma\left(h^\sharp, \rho^\sharp\right) = \left\{ (h, \rho) \;\middle|\; \begin{array}{l} h \in \gamma^{\mathrm{Heap}}(h^\sharp) \\ \rho \in \gamma_h^{\mathrm{Env}}(\rho^\sharp) \end{array} \right\} \subseteq \mathrm{Heap} \times \mathrm{Env}$$

At the memory level, each transfer function will be of the form

$$\begin{array}{rll} \mathcal{F} : & \mathrm{Heap} \times \mathrm{Env} \to & \mathrm{Heap} \times \mathrm{Env} \\ & (h, \rho) & \mapsto (f(h, \rho), g(h, \rho)) \end{array}$$

To correctly abstract a transfer function, we have to propose a function $\mathcal{F}^\sharp$ looking like

$$\begin{array}{rll} \mathcal{F}^\sharp : & \mathrm{Heap}^\sharp \times \mathrm{Env}^\sharp \to & \mathrm{Heap}^\sharp \times \mathrm{Env}^\sharp \\ & (h^\sharp, \rho^\sharp) & \mapsto \left(f^\sharp(h^\sharp, \rho^\sharp), g^\sharp(h^\sharp, \rho^\sharp)\right) \end{array}$$

and verifying the following "classical" correctness criterion

$$\begin{array}{l} \forall (h^\sharp, \rho^\sharp) \in \mathrm{Heap}^\sharp \times \mathrm{Env}^\sharp, \\ \quad \left\{ (f(h, \rho), g(h, \rho)) \;\middle|\; \begin{array}{l} h \in \gamma^{\mathrm{Heap}}(h^\sharp) \\ \rho \in \gamma_h^{\mathrm{Env}}(\rho^\sharp) \end{array} \right\} \subseteq \left\{ (h', \rho') \;\middle|\; \begin{array}{l} h' \in \gamma^{\mathrm{Heap}}(f^\sharp(h^\sharp, \rho^\sharp)) \\ \rho' \in \gamma_{h'}^{\mathrm{Env}}(g^\sharp(h^\sharp, \rho^\sharp)) \end{array} \right\} \end{array}$$

This criterion can be equivalently reduced to the conjunction of two criteria

$$\begin{array}{l} \forall (h^\sharp, \rho^\sharp) \in \mathrm{Heap}^\sharp \times \mathrm{Env}^\sharp, \\ \forall (h, \rho) \in \gamma^{\mathrm{Heap}}(h^\sharp) \times \gamma_h^{\mathrm{Env}}(\rho^\sharp), \quad f(h, \rho) \in \gamma^{\mathrm{Heap}}(f^\sharp(h^\sharp, \rho^\sharp)) \end{array} \tag{1}$$

$$\begin{array}{l} \forall (h^\sharp, \rho^\sharp) \in \mathrm{Heap}^\sharp \times \mathrm{Env}^\sharp, \\ \forall (h, \rho) \in \gamma^{\mathrm{Heap}}(h^\sharp) \times \gamma_h^{\mathrm{Env}}(\rho^\sharp), \quad g(h, \rho) \in \gamma_{f(h,\rho)}^{\mathrm{Env}}(g^\sharp(h^\sharp, \rho^\sharp)) \end{array} \tag{2}$$

Contrary to the criterion (1), the criterion (2) is problematic because it contains two distinct instances γ_h^{Env} and $\gamma_{f(h,\rho)}^{\mathrm{Env}}$. As we have seen previously, properties produced by combining generic connections and parameterised concretizations only contain a single context element. So we can not prove (2) by only combining this kind of properties.

We can however, reduce the proof of (2) into two sufficient (but not necessary) conditions, one dealing with f, the other with g:

$$\begin{array}{l} \forall (h^\sharp, \rho^\sharp) \in \mathrm{Heap}^\sharp \times \mathrm{Env}^\sharp, \\ \forall (h, \rho) \in \gamma^{\mathrm{Heap}}(h^\sharp) \times \gamma_h^{\mathrm{Env}}(\rho^\sharp), \quad g(h, \rho) \in \gamma_h^{\mathrm{Env}}(g^\sharp(h^\sharp, \rho^\sharp)) \end{array} \tag{3}$$

$$\forall (h, \rho) \in \mathrm{Heap} \times \mathrm{Env}, \ \gamma_h^{\mathrm{Env}} \mathrel{\dot\subseteq} \gamma_{f(h,\rho)}^{\mathrm{Env}} \tag{4}$$

The criterion (3) now only contains a single instance γ_h^{Env} of the environment concretization (contrary to (2)) and is well-suited to be proved by combining properties given by some generic connection constructors.

The criterion (4) remains nevertheless problematic because like in (2), several instance of γ^{Env} appear. The next section will be dedicated to this criteria. We will propose a slightly change in the generic environment connection definition which will allow us to prove (4) in a modular way without making appear a context notion in the definition.

5 Concretization functors

The improvement we will make in generic connection definition will be based on *concretization functionals* : operators which transform concretizations into other concretizations.

5.1 Example and definition

An example of such operator has already been seen in lemma 1.

$$\begin{array}{rcl} \Gamma : \left(\left(\mathrm{Val}^\sharp, \sqsubseteq_{\mathrm{Val}^\sharp}\right) \to_m (\mathcal{P}(\mathrm{Val}), \subseteq)\right) & \to & \left(\left(\mathrm{Env}^\sharp, \sqsubseteq_{\mathrm{Env}^\sharp}\right) \to_m (\mathcal{P}(\mathrm{Env}), \subseteq)\right) \\ \gamma^{\mathrm{Val}} & \mapsto & \rho^\sharp \mapsto \left\{\rho \;\middle|\; \forall x \in \mathrm{Var},\ \rho(x) \in \gamma(\rho^\sharp(x))\right\} \end{array}$$

This kind of operator is under-lying in many generic construction found in the abstract interpretation literature. A natural condition we could impose on such operator is *monotonie*, hence obtaining *concretization functors*.

Definition 2. *(**Concretization functor**) Given four partially ordered sets $(A, \sqsubseteq_A)$, $\left(A^\sharp, \sqsubseteq_{A^\sharp}\right)$, $(B, \sqsubseteq_B)$ and $\left(B^\sharp, \sqsubseteq_{B^\sharp}\right)$, a* concretization functor *is an operator Γ taken in $\left(\left(A^\sharp, \sqsubseteq_{A^\sharp}\right) \to_m (A, \sqsubseteq_A)\right) \to \left(\left(B^\sharp, \sqsubseteq_{B^\sharp}\right) \to_m (B, \sqsubseteq_B)\right)$ which verifies a monotony property like*

$$\forall \gamma_1, \gamma_2 \in \left(\left(A^\sharp, \sqsubseteq_{A^\sharp}\right) \to_m (A, \sqsubseteq_A)\right), \quad \gamma_1 \ \dot{\sqsubseteq}_A \ \gamma_2 \quad \Rightarrow \quad \Gamma(\gamma_1) \ \dot{\sqsubseteq}_B \ \Gamma(\gamma_2)$$

A concretization functor is hence preserving relative precision between concretizations. This monotony property appears to be very natural and satisfied by many concretization operators found in the literature (see the generic construction of weak relational environment in [11] for a good example). As far a we know this property has never been explicitly used or noticed.

This notion will now be integrated in a new definition of generic environment connection.

Definition 3. *(**Revisited *generic environment connection***) We call* generic environment connection *a functional which associates with any partially ordered set $(\mathcal{P}(Val), \subseteq)$ a 5-upplet $\left(Env^\sharp, \sqsubseteq_{Env^\sharp}, \Gamma^{Env}, \mathrm{get}^\sharp, \mathrm{subst}^\sharp\right)$ with*

- $\left(Env^\sharp, \sqsubseteq_{Env^\sharp}\right)$ *a partially ordered set*
- $\Gamma \in \left(Val^\sharp \to_m \mathcal{P}(Val)\right) \to \left(Env^\sharp \to_m \mathcal{P}(Var \to Val)\right)$ *a concretization functor*
- $\mathrm{get}^\sharp \in Env^\sharp \times Var \to Val^\sharp$ *a correct approximation of the function giving the value attached with each variable*

$$\begin{array}{l} \forall \gamma \in \left(Val^\sharp \to_m \mathcal{P}(Val)\right), \\ \forall \rho^\sharp \in Env^\sharp,\ \forall x \in Var, \quad \left\{\rho(x) \;\middle|\; \rho \in \Gamma^{Env}(\gamma)\left(Env^\sharp\right)\right\} \ \subseteq \ \gamma(\mathrm{get}^\sharp(\rho^\sharp, x)) \end{array}$$

– $\mathrm{subst}^\sharp \in Env^\sharp \times Var \times Val^\sharp \to Env^\sharp$ *a correct approximation of the function which substitute a value with an other one in a variable*

$$\begin{array}{l}\forall \gamma \in \left(Val^\sharp \to_m \mathcal{P}(Val) \right), \\ \forall \rho^\sharp \in \rho^\sharp,\ \forall x \in Var,\ \forall v^\sharp \in Val^\sharp, \\ \qquad \left\{ \rho[x \mapsto v] \;\middle|\; \begin{array}{l} \rho \in \Gamma^{Env}(\gamma)(\rho^\sharp) \\ v \in \gamma(v^\sharp) \end{array} \right\} \subseteq \Gamma^{Env}(\gamma)(\mathrm{subst}^\sharp(\rho^\sharp, x, v^\sharp)) \end{array}$$

The modification used here is made at the level of the concretization function which is no more fixed but now parameterised by any value concretization. Concerning primitive abstract operator $\mathrm{get}^\sharp$ and $\mathrm{subst}^\sharp$, the quantification made on all value concretization does not required more proof than in the previous definition because γ_{Val} was already anonymous. We can hence affirm that this new definition is not more restrictive or specialized than the previous: only the monotony property of Γ has been added and it is a very natural property which do not restrict the generic construction we can use.

We will now explain why these generic connection enable us to prove (4) in a modular fashion.

5.2 Using the functorial property in proof

With our new definition of generic environment connection, the concretization γ^{Env} used in the example of Section 4 is now of the form

$$\gamma^{\mathrm{Env}} = \Gamma^{\mathrm{Env}}\left(\gamma^{\mathrm{Val}}\right)$$

Hence the criterion (4) can now be reduced to a property on γ^{Val}.

Lemma 2. *If $\gamma^{Env} = \Gamma^{Env}\left(\gamma^{Val}\right)$ with Γ^{Env} a concretization functor, then the criterion*

$$\forall (h, \rho) \in Heap \times Env,\ \gamma_h^{Val} \mathrel{\dot{\subseteq}} \gamma_{f(h,\rho)}^{Val} \tag{5}$$

implies

$$\forall (h, \rho) \in Heap \times Env,\ \gamma_h^{Env} \mathrel{\dot{\subseteq}} \gamma_{f(h,\rho)}^{Env}$$

Proof. It is a direct consequence of the monotony property of Γ^{Env}.

The remaining proof condition (5) is thus structurally smaller: it now deals with value abstraction. It can be seen has a *conservative* requirement. The concretization associated with the transformation of the heap h should contain all the properties of the original one. It is a strong property but the generic connection definition allow us to move it at the level of the value connection without sacrificing the genericity of the environment connection.

It remains us to explain how such proof can be managed at the level of the value abstraction.

5.3 Establishing the conservative requirement

In the context of a full correctness proof there will be as many proof condition like (5) as function f encountered in the different transfer functions of the language. We propose to factorize these proofs by cutting such conditions into two new conditions. This cut is done by introducing a well-chosen pre-order on the context domain.

We will need to introduce a notion of *monotone parameterised concretization.*

Definition 4. ***(monotone parameterised concretization)*** *Given a pre-order relation $\preceq_C \subseteq C \times C$ on a set C, a parameterised concretization $\gamma \in C \to D^\sharp \to \mathcal{P}(D)$ is* monotonously parameterised *with respect to $\preceq_C$ if*

$$\forall (c_1, c_2) \in C^2,\ c_1 \preceq_C c_2 \ \Rightarrow\ \gamma_{c_1} \dot{\subseteq} \gamma_{c_2}$$

Lemma 3. *Let $\mathcal{S} \subseteq (Heap \times Env) \to Heap$ be a set of function. Let γ^{Val} be a parameterised value concretization and $\preceq_{Heap}$ a pre-order on Heap. If*

$$\gamma^{Val} \textit{ is monotone with respect to } \preceq_{Heap} \tag{6}$$

and

$$\forall f \in \mathcal{S},\ \forall (h, \rho) \in Heap \times Env,\ h \preceq_{Heap} f(h, \rho) \tag{7}$$

then

$$\forall f \in \mathcal{S},\ \forall (h, \rho) \in Heap \times Val,\ \gamma_h^{Val} \dot{\subseteq} \gamma_{f(h,\rho)}^{Val}$$

As we explain before, this proof method is not applicable for all parameterised value concretization and all programming language. The main restriction is on the existence of a well-suited pre-order on context domain.

This existence is nevertheless ensured by all the previous abstraction example we gave previously in Section 3 and 4:

- For example 1, we can take

  $$\preceq_{\text{Heap}} = \left\{ (h_1, h_2) \;\middle|\; \begin{array}{l} \text{dom}(h_1) \subseteq \text{dom}(h_2) \\ \forall \text{loc} \in \text{dom}(h_1),\ \text{class}(h_1(\text{loc})) = \text{class}(h_2(\text{loc})) \end{array} \right\}$$

  The value concretization chosen in this example is then monotone with respect to this pre-order because if h_1 and h_2 are heap verifying $h_1 \preceq_{\text{Heap}} h_2$, if loc belongs to $\gamma_{h_1}(s)$ then loc $\in \text{dom}(h_1)$ and $\text{class}(h_1(\text{loc})) \in s$. But $\text{dom}(h_1) \subseteq \text{dom}(h_2)$, so loc $\in \text{dom}(h_2)$ and because $\text{class}(h_1(\text{loc})) = \text{class}(h_2(\text{loc}))$, we can affirm that $\text{class}(h_2(\text{loc})) \in s$. We then have demonstrated that loc $\in \gamma_{h_2}(s)$.
  The property 7 will be verified by any transfert function which does not remove objects in the heap, neither modify their class. It is effectively the case for all transfer function of programming language like Java or bytecode Java without dealing with garbage collector[1].

[1] Dealing with garbage collection could be done by restricting value to accessible values from the variable in the environment. It would certainly complicate the proof and we have not yet explored this eventuality.

- The same pre-order as before can be used to deal with example 2.
- For example 3, the context is no more a heap but a partial trace. The relation $\preceq_{\text{Trace}}$ is thus sufficient :

$$\preceq_{\text{Trace}} = \{(tr_1, tr_2) \mid tr_1 \text{ is a prefix of } tr_2\}$$

  Indeed, if tr_1 is a partial trace prefix of a partial trace tr_2, all allocations made in tr_1 appear in tr_2. Thus the monotonicity of γ^{Val} with respect to $\preceq_{\text{Trace}}$ is proved.
  For the criterion 7, we only have to verify that all transfer function only put new states on previous partial trace, which is indeed the case.

5.4 Summarizing the proof method

We now summarize our proof method for establishing the correctness of the function $\mathcal{F}^\sharp$ with respect to $\mathcal{F}$ (example taken in Subsection 4.2)

- The correctness criterion is split into two equivalent criteria (1) and (2). (1) leads to modular proofs because it relies on the same parameterised concretization, but (2) is not.
- The criterion (2) is then split into two sufficient criteria (3) and (4). (3) is provable using generic connection constructions.
- To establish (4) we introduce a notion of concretization functor and a well chosen pre-order. (4) is hence split into criteria (6) and (7). (6) only deal with the abstraction made on values. (7) is a proof about the semantic of the language.

6 Modular Machine checked proof of a bytecode analyser

This proof technique has been experimented for proving the correctness of a generic Carmel static analyser. This analysis computes a state invariant for each program point. The abstract state is thus of the form

$$\text{State}^\sharp = \text{ProgPoint} \to \left(\text{Heap}^\sharp \times \text{LocalVar}^\sharp \times \text{Stack}^\sharp\right)$$

with $\text{Heap}^\sharp$, $\text{LocalVar}^\sharp$ and $\text{Stack}^\sharp$ generic abstract domains for heap, local variables and operand stack abstraction.

The generic static analyser is parameterised by five generic connections (for values, operand stacks, local variables, objects and heaps) and two base abstractions (a parameterised one for locations and a classical simple one for integers). Figure 3 shows the Coq interface definition for the operand stack. The interface is parameterised by a lattice structure PV on a set V (the lattice of abstract values). The interface contains 12 elements. First, the set t of abstract stacks, the lattice structure Pos on t, and the concretization functor gamma which takes concretization between $\mathcal{P}(\text{Val})$ and PV and returns a concretization between $\mathcal{P}(\text{Stack})$ and Pos. The monotonicity property of gamma is required by the field gamma_monotone. At last, nil_ab, pop_ab, top_ab and push_ab are four basic abstract operators of the stack domain with their corresponding correctness properties. (Post pop_op) represents the post operator applied on the relation pop_op. This interface and the others (for local variables, objects, ...) are collected in the file AlgebraType available on-line for the interested reader.

```
Record OperandStackConnection (V:Set) (PV:Lattice V) : Type := {
  t : Set;
  Pos : Lattice t;
  gamma :  Gamma (PowPoset Value) PV → Gamma (PowPoset OperandStack) Pos;
  gamma_monotone : ∀g1 g2,
    orderGamma g1 g2 → orderGamma (gamma g1) (gamma g2);

  nil_ab : t;
  nil_ab_correct : ∀g, (λs. s = nil) ⊆ (gamma g nil_ab);

  pop_ab : t → t;
  pop_ab_correct : ∀g s, ((Post pop_op) (gamma g s)) ⊆ (gamma g (pop_ab s));

  top_ab : t → V;
  top_ab_correct : ∀g s, ((Post top_op) (gamma g s)) ⊆ (g (top_ab s));

  push_ab : V → t → t;
 push_ab_correct : ∀g v s, ((Post2 push_op) (g v) (gamma g s)) ⊆ (gamma g (push_ab v s))
}.
```

Fig. 3. Operand Stack connection interface

The correctness of the analysis is established for any correct integer, reference, value, operand stack, local variables, object and heap connection. We have implemented various instanciations of the different interfaces

- integers : abstraction by type (only one element in the abstract domain) and constant abstraction,
- references : abstraction by class (example 1) and abstraction by creation point (example 2),
- values : abstraction by sum of the reference and the numeric abstraction with two possibilities for the representation of the null constant (represented by the bottom element or by a specific abstract object)
- stacks, local variables, objects : structural abstraction (structure is preserved) or one abstract value to abstract all the elements of the data
- heaps : only one instantiation parameterised by any object abstraction and reference abstraction (with some restriction on the lattice used for references)

Compared with the previous proof done in [3], we have hence two important improvements. First, the proof is now modular, abstractions on semantic subdomains can be changed without redoing all the global proof : this is important for incremental development and maintaining of the proof. Second, each subdomain abstraction is generic and independent from the others abstractions, which helps considerably during the proof development by splitting the global proof into several simpler proofs.

7 Related works

In a previous paper [3], we have shown how to formalise a constraint-based data flow analysis in the specification language of the Coq proof assistant. We proposed a library of lattice functors for modular construction of complex abstract domains. Constraints were expressed in an intermediate representation that allowed for both efficient constraint resolution and correctness proof of the analysis with respect to an operational semantics. The proof of existence of a correct, minimal solution to the constraints was constructive which means that the extraction mechanism of Coq provided a provably correct data flow analyser in Ocaml[12]. The library of lattices together with the intermediate representation of constraints were defined in an analysis-independent fashion that provides a basis for a generic framework for proving and extracting static analysers in Coq. Nevertheless, no specific methodology was proposed to handle the correctness proof of the abstract semantic with respect to the standard semantics.

The majority of mechanical verifications of program analyses have dealt with the Java byte code verifier. Bertot [2] used the Coq system to extract a certified bytecode analyser specialized for object initialization. Barthe *et al.* [1] have shown how to formalise the Java Card byte code verification in the proof assistant Coq by isolating the byte code verification in an executable semantics of the language. Klein and Nipkow [9] have proved the correctness of a Java byte code verifier using the proof assistant Isabelle/HOL. All these works do not rely on a general theory of static analysis like abstract interpretation, and are oriented towards type verification.

The notion of parameterised concretization function has been implicit in several works and was made explicit in the thesis of Isabelle Pollet [13]. In this work, abstract interpretation of Java program are presented with the help of parameterised concretization functions which are used to relate concrete and abstract values with respect to a relation between locations. However, to the best of our knowledge, no one has identified the *functor property* presented here which is essential for the modularization and mechanization of the proofs.

Concerning proof modularity, only a few work propose a so modular approach than us. The main reason is that research paper rarely deal with a deep hierarchical semantic domain. In our context, splitting the proof development following the semantic hierarchy was useful, specially to machine-checked the proof. Much works are dedicated to propose one single powerful construction of abstract domain parameterised by some base domain, see for example works of Miné [11] or Cortesi *and al* [6]. But base abstraction are not parameterised (because the target analyses do not need this notion) and thus they did not encounter the same technical problem as us. In [13] several generic connection constructor are given to analyse heap structure and a common interface is proposed. Nevertheless this interface makes an explicit use of the parameter : what we try to avoid with our notion of concretization functor. But the proposed constructor allow more powerful analyses than those we implement in Coq. A last interesting related work can be found in the course note of Patrick Cousot [7] where the abstract interpreter construction is modularized following each semantic sub-domain. But once again, no parameterised abstraction is used then our functor notion is not required.

8 Conclusion

Mechanised correctness proofs of static analyses for realistic programming languages requires proof principles for simplifying the proof development. Like in other software engineering activities, modular correctness proofs are desirable because they are easier to develop and to maintain. We observe that one obstacle to modularity is due to the fact that the concrete state is a complex object with many apparently inter-related components. The abstract domain has to reflect these relations but using a full-fledged abstract domain with standard (relational) concretisations leads to proofs with poor modular structure. In this paper we have shown how parameterised concretisation functions forms a basis for proof principles that allow to capture the necessary relational information while using concretisation functions as if we were working with non-relational domains.

To arrive at these proof principles, we have extended the theory of parameterised concretisations with the key notion of concretisation functors (inspired by the notion of functors in *e.g.*, OCAML) that make explicit the compositional way in which concretisation functions for complex domains are constructed from concretisation of their simpler constituents. We have formulated and proved an important property of concretisation functors that shows how a properly chosen pre-order on the concrete domains can greatly simplify the correctness proof for a large class of transfer functions.

The motivation for these theoretical developments came from a mechanised correctness proof for a generic static analysis for stack-based byte code language with memory allocation (not entirely unlike Java Card). As argued in Section 6 the proof principles have demonstrated their practical value by reducing the proof effort considerably. We tested this genericity by instantiating the abstract domain for memory references with two well-known abstractions while keeping the rest of the abstract state fixed. This was a non-trivial task because these reference abstraction use distinct parameterisations.

We now dispose of a proof technique which allows to certify complex static analysis for real languages in a reasonable time. A further work could be to achieve such an analysis for a byte code languages with all the features of the Java Card languages (exception, array, virtual calls). Propose certified analyser implementation without loosing efficiency require still works when dealing with complex abstraction.

References

1. Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In *Proc. ESOP'01*, number 2028 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
2. Yves Bertot. Formalizing a JVML Verifier for Initialization in a Theorem Prover. In *Proc. CAV'01*, number 2102 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
3. David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. ESOP'04*, number 2986 in Lecture Notes in Computer Science, pages 385–400. Springer-Verlag, 2004.
4. David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 2005. Extended version of [3], to appear.
5. The Coq Proof Assistant. `http://coq.inria.fr/`.
6. Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *POPL*, pages 227–239, 1994.
7. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
9. Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.
10. Renaud Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005, Trusted Logic SA, May 2001.
11. A. Miné. A few graph-based relational numerical abstract domains. In *SAS'02*, volume 2477 of *LNCS*, pages 117–132. Springer-Verlag, 2002.
12. The Objective Caml language. `http://caml.inria.fr/`.
13. Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java*. PhD thesis, Université catholique de Louvain, Belgium, 2004.
14. Igor Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Logic and Automated Reasoning*, 2004. To appear.