Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Automatic Coq Proofs Generation from Static Analyzers by Lightweight Instrumentation

Manuel Garnacho, Michaël Périn

Verimag Research Report n^o TR-2011-18

January 2011

Automatic Coq Proofs Generation from Static Analyzers by Lightweight Instrumentation

Manuel Garnacho, Michaël Périn

January 2011

Abstract

This paper deals with program verification and more precisely with the question of how to provide verifiable evidence that a program verifies certain semantics properties. Program processing tools such as compiler or static analyzers are complex pieces of software which may contain errors. The idea of using analyzers as guessing algorithms and proving the discovered properties by independent means has been proposed a decade ago. However, automatically generating the proofs without user interaction is still a major challenge. We present a methodology of instrumentation of existing static analyzers based on abstract interpretation to make them produce certificates of their results. We apply our methodology on an existing static analyzer that discovers invariants of array-processing programs which can be expressed in first-order logic. Certificates are provided as COQ proofs based on Floyd-Hoare's method for proving program invariants.

Keywords: certification, static analyzer, instrumentation, coq proof

Reviewers: accepted to IWIL'2010

Notes: The authors had no fund to join the conference in Yogyakarta (Indonesia), the paper was therefore retracted.

How to cite this report:

```
@techreport { ,
title = { Automatic Coq Proofs Generation from Static Analyzers by Lightweight Instrumentation},
authors = { Manuel Garnacho, Michaël Périn},
institution = {  Verimag Research Report },
number = {TR-2011-18},
year = { },
note = { }
}
```

1 Certifying the verdict of validation tools

Program processing tools (compilers, validation tools) are the cornerstones on which the safety of software is built. At the same time these tools are based on subtle algorithms and complex theoretical background. This raises the problem of proving the correctness of the implementation of such a tool. This is hard to achieve in practice. Therefore, in a certification process the validation tool is assumed unsafe and its verdict is considered no more trustable than a "guess". A high-level of confidence in the accuracy of the "guess" can be reached by providing a certificate for each particular verdict of the tool and checking it using a trusted certificate checker [23, 17]. Clearly, we don't want to build the certificate by hand for each run of the tool ; the goal is to extend the validation tool so that it produces the certificate by instrumentation of its computations.

1.1 Automatic generation of proofs by instrumentation

In this paper we consider certificates in the form of foundational proofs for which exist trustable proof-checkers (*e.g.* LF, COQ, HOL) [1, 19]. It is very unlikely that when provided with the "guess", a standard theorem prover will automatically produce a proof of the verdict: the strategy of the validation tool would have to be provided to the theorem provers as proof tactics which means re-implementing a large part of the validation tool in the prover. Moreover, verdicts of a validation tool can be considered as certificates if the validation tool has been certified. Besson, Jensen and Pichardie [4] have proposed to use the fixpoint generated by a certified abstract interpreter as a certificate. Unfortunately their approach based on certified abstract interpretation which is a technique for extracting a static analyzer from the constructive proof of its soundness doesn't work on existing tools implemented with standard programming languages.

We investigate a more practical approach that consists in instrumenting an existing tool so that it automatically builds a certificate of its verdict in the form of a foundational proof. To do so, we distinguish functions that implement the *heuristics* of the tool (they are responsible for precision, termination) from the *semantic functions* which are accountable for correctness. Only latter need to be instrumented to generate proof-terms.

An instrumented function is said to be *certifying* (instead of *certified*) as it generates proof-terms to justify its computations but there is no guarantee that it will produce a valid proof-term for all possible inputs. The term *certified* is reserved for functions whose correctness has been established on all their input domain.

The *semantic function* of the tools are instrumented as follows: First, the correctness of a function f is stated as a property relating its input and its result. It corresponds to a (partial) specification Q of the function f as pre and $post$ conditions. Second, a *certifying function*, denoted by f^π, is created that must return *justified results* in the form of a pair made of the result of the original function f and a proof that this result meets the specification. Intuitively, a certifying function builds a specialization of a general correctness proof for the particular entry of the call using a proof pattern. The general form of a certifying function is $\mathsf{f}^\pi(x) \stackrel{def}{=} \mathsf{let}\ r = \mathsf{f}(x)\ \mathsf{in}\ (r, \pi_{pre(x)\Rightarrow post(r)})$. We use π_f to denote proof patterns, π to denote the corresponding proof-terms generated at execution and π_φ to point out that it is a proof of the property φ.

Next, each computation $r := \mathsf{f}(i)$ is replaced by $(r, \pi) := \mathsf{f}^\pi(i)$. When the computation of a function f involves results of other function calls, the associated proof-term π_φ is built from the proofs $\pi_{\varphi_1}, \ldots, \pi_{\varphi_n}$ provided by the call of certifying functions. All the instrumentation efforts of a function then lie in combining the subproofs using appropriate rules of the deduction system to demonstrate φ from $\varphi_1 \wedge \ldots \wedge \varphi_n$.

This can hardly be achieved for large and complex functions such as the abstract transfer function which represents a great amount of code of an analyzer. In general it contains many case distinctions in order to recognize the situation where a very specific treatment can increase the precision of the analysis. In such case, it is hard to find a general proof pattern in this large collection of tricky situations. Therefore, instead of generating a proof that follows the computations we use weakest precondition calculus to obtain a logical implication in first order logic. Then, we prove that implication using the certifying version of the abstraction mapping α and the certifying version of the comparison operator $\sqsubseteq$ of the abstract lattice.

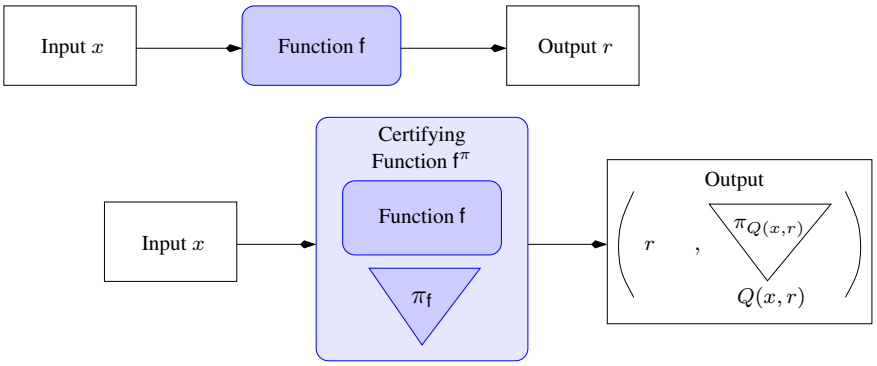

Figure 1: instrumentation of the function f using the proof pattern π_{f}

1.2 Experimentation on an existing analyzer

We conduct an experimentation on a static analyzer, ENKIDU [16] which discovers invariant properties of array-processing programs. Running ENKIDU on the *insertion-sort* algorithm below, the analyzer automatically associates an inductive assertion to each program location. The discovered assertion Φ_9 on the exit point says that the array $A[2..n]$ is sorted, that is, $\forall \ell,\ 2 < \ell \leq n \Rightarrow A[\ell - 1] \leq A[\ell]$.

```
1:i := 2 ;
2:while(i ≤ n) do
3: v := A[i] ; j := i ;
4: while(1 < j ∧ v < A[j − 1]) do
5:    A[j] := A[j − 1] ; j := j − 1
6: od        Φ6 =def {A[j − 1] ≤ v ⋀ ∀ℓ, 1 < ℓ ≤ j − 1 ⇒ A[ℓ − 1] < A[ℓ]}
7: A[j] := v ;
          Φ7 =def {∀ℓ, 1 < ℓ ≤ j ⇒ A[ℓ − 1] < A[ℓ]}
8: i := i + 1
9:od        Φ9 =def {∀ℓ, 1 < ℓ ≤ n ⇒ A[ℓ − 1] ≤ A[ℓ]}
```

The justified version of the analyzer automatically generates the proofs of the discovered assertions in the COQ syntax. These proofs are conducted in three steps: the pre- and post conditions of an instruction are combined into a Hoare triple, for instance $\{\Phi_6\}\ A[j] := v\ \{\Phi_7\}$ for the assignment at line 7. Next, using weakest precondition calculus, the triple is then changed into an equivalent logical formula, $\Phi_6 \Rightarrow wp(A[j] := v, \Phi_7)$, which is then established using the proof-term generated by the analyzer while processing the assignment at line 7.

Certifying array-processing programs involve dealing with array aliases. The naive definition of wp for assignment, $wp(A[j] := v, \Phi) \stackrel{def}{=} \Phi[_{A[j]/v}]$, is unsound and incomplete for programs dealing with objects, pointers or arrays, as noticed in [6]. For instance, assume that the assertion Φ bears on a cell $A[\ell]$ while the cell $A[j]$ is assigned, and $\ell = j$, then the syntactic substitution ignores the alias between $A[j]$ and $A[\ell]$. To address that issue, [6] that represents the array A as a function f_A from indices to values. Then, the assignment $A[j] := v$ updates the function f_A which becomes $(\lambda i.\ \mathsf{if}\ (i = j)\ \mathsf{then}\ v\ \mathsf{else}\ A[i])$ and the Hoare substitution does not apply to the cell $A[j]$ but to the array A: $\{\Phi[_{A/f_A}]\}\ A[j] := v\ \{\Phi\}$ This idea can be rephrased using the $store/select$ axioms for arrays [22]. We implemented a similar solution proposed that avoids dealing with λ-abstraction and $store/select$ axioms. It consists in expliciting all the potential aliases in the assertion according to the assigned array cell, prior to the substitution. In the example, the subterms $A[\ell - 1]$ and $A[\ell]$ of Φ_7 can potentially be in alias with the assigned cell $A[j]$. Our weakest precondition calculus produces the formula below where the aliases with $A[j]$ are made explicit:

$$wp_{A[j]:=v}(\Phi_7) = \forall \ell,\ 1 \leq \ell \leq j \Rightarrow \bigwedge \begin{array}{l} \ell \neq j \Rightarrow \left(\bigwedge \begin{array}{l} \ell - 1 \neq j \Rightarrow A[\ell - 1] \leq A[\ell] \\ \ell - 1 = j \Rightarrow A[j] \leq A[j+1] \end{array} \right) \\ \ell = j \Rightarrow \left(\bigwedge \begin{array}{l} j - 1 \neq j \Rightarrow A[j-1] \leq A[j] \\ j - 1 = j \Rightarrow A[j] \leq A[j] \end{array} \right) \end{array}$$

This formula is equivalent to:

$$wp(A[j] := v, \Phi_7) = \forall \ell,\ \bigwedge \left\{ \begin{array}{ll} 1 \leq \ell \leq j \wedge \ell \neq j \wedge \ell - 1 \neq j \Rightarrow A[\ell - 1] \leq A[\ell] & (1) \\ 1 \leq \ell \leq j \wedge \ell \neq j \wedge \ell - 1 = j \Rightarrow A[j] \leq A[j+1] & (2) \\ 1 \leq \ell \leq j \wedge \ell = j \wedge j - 1 \neq j \Rightarrow A[j-1] \leq A[j] & (3) \\ 1 \leq \ell \leq j \wedge \ell = j \wedge j - 1 = j \Rightarrow A[j] \leq A[j] & (4) \end{array} \right.$$

The validity of the property discovered by ENKIDU are established with respect to the semantics of guards and assignments defined by the wp calculus. No user-interaction is required: when the analyzer discovers a valid property, the instrumented version produces a proof of it.

1.3 Contributions

Our instrumentation method can be applied to a large class of static analyzers based on *fixpoint computation* and *abstraction*. We show that when an untrusted tool produces a valid verdict then the instrumentation produces a valid proof of the verdict. We also address the challenge of producing *foundational proofs* for trusted proof-checkers where each evidence must be provided in terms of very basic deduction rules (or lemmata themselves based on these rules).

Inspired by initial works of Pnueli [26] and Necula [25] in the area of compilation, then following previous works about proof-producing static analysis [7, 27], we investigate a novel instrumentation technique that (1) applies to existing static analyzers which have not been developed with the goal of producing proofs ; (2) that uses proof patterns instead of proof searching algorithm. The whole proof is produced automatically as the combination – directed by computations – of these basic justifications. And (3) as far as we know, our instrumentation is the lightest one that automatically produces certificates from a fixpoint computation: Indeed, it avoids the instrumentation of the abstract transfer function and therefore it is robust to changes in the heuritics of the analyzer. Instead the instrumentation effort is concentrated on the operators of the abstract domains that are usually implemented as libraries shared among several analysis.

1.4 Related work

Ideally, one would like to be provided with a proof of correctness of the validation tool. If the verification algorithm was proved in the COQ proof assistant, then the implementation of the validation tool could be obtained by extraction from the proof. This specific COQ feature guarantees a tool implementation correct by extraction. The WHY tool and its successors [13] for the analysis of imperative programs implement a calculus of the weakest precondition in COQ. It is proved correct with respect to an operational semantics but he does not provide an automatic technique to generate COQ certificates. These certificates are required since WHY uses uncertified external tools to annoted programs. This extraction approach is also promoted in [4] for the development of static analysis using abstract interpretation. Unfortunately, it does not provide a solution for the many valuable existing tools not developed in COQ. Therefore, in the case of an existing analyzer, instrumentation seems a more practical approach. In contrast to [4], instrumentation does not produce a certified analyzer which has been proved once and for all. Instead it generates a dedicated proof for each run of the analyzer.

The idea of adding certificates to verdicts of validation tools was introduced by Namjoshi [23] for model-checkers of temporal formulæ. The basis for producing evidence is provided (invariants for safety properties and ranking functions for liveness properties). Certificates take the form of deductive proofs in a high-level proof-system where uninteresting deduction steps are skipped. Other forms of certificates have been proposed for model-checking [17, 28]. Meanwhile it has been recognized in the proof-carrying code community that foundational proof-checkers are more reliable [2] than type-specialized checkers. In the foundational framework, a proof-term is a combination of deduction rules from higher-order logic. Proof-checking then consists of a recursive traversal of the proof-term to check that each step obeys the syntactic constraint of the deduction schemes of the proof system. The LF and COQ foundational proof-systems are

based on a few deduction rules and come with very small, simple and trusted[1] proof-checkers [1, 3]. Such proof-checkers provide a high-level of confidence. A challenge is to automatically produce foundational proofs from validation tool. Indeed, each evidence must be provided in terms of very basic deduction rules (or lemmata based on these rules). In a similar philosophy as PCC [24], our approach, by generating proofs from a static verification performed on the untrusted code, can also be used to provide guarantees to the user of dowloaded code.

Works closely related to ours [7, 27] provide syntax directed strategies to prove the correctness of the result of an abstract interpretation. This approach stricly follows computations of static analyzers and some part of that are very complicated and not realistic in practice (as instrumenting the abstract transfer functions). The final proof also refers to the abstract domain and exploits some property of the concretization function. Hence, the proof must maintain a relation between the logic of the proof-checker and the abstract domain, meaning that a translation of each abstract value into the logic of the proof-checker must be provided and proved correct. Our justification technique avoids some of these difficult part and greatly reduces the instrumentation work: the proof only refers to the original program and assertions expressed in first order logic; it does not mention the abstract domain neither the abstract semantics. This is an advantage for a certification process where the evaluators must understand and agree on the semantics axiom.

Overview. The remainder of this paper is structured as follows: Section 2 recalls the principle of abstract interpretation and presents the typical architecture of a static analyzer. Then, Section 3 is dedicated to the instrumentation of a generic analyzer with proof patterns so that it produces justifications, in the COQ syntax, of the properties discovered by the abstract interpretation. We apply our technique to an existing static analyzer (ENKIDU [16]) in section 4: we detail the instrumentation of some operators of the abstract domain and report experimental results. Finally, we draw a conclusion and present some directions for future work in section 5.

2 Architecture of a static analyzer

For the needs of the following presentation, we start with a brief summary of what is common to a large class of static analyzers: the principle of abstract interpretation and fixpoint computation [10].

Without loss of generality, a program can be seen as a control flow graph (*i.e.* a transition system) where each edge (*i.e.* a transition $\tau_{qq'}$) between two program locations q and q' is annotated by a guard g, or an assignment $v := e$. A program state is a couple (q, σ) made of a program point q and a mapping σ of variables to values, which represents a memory state. The goal of a static analysis is to associate an assertion (*i.e.* an invariant property) to each node of the program. Invariant properties are not computable in general, thus, a static analyzer focuses on a well-chosen class of properties for which invariants can be obtained by fixpoint computation on an abstract lattice $(\mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The abstract domain $\mathcal{A}$ represents the target class of properties. It has a least and a greatest element (resp. $\bot$ and $\top$), a partial order $\sqsubseteq$ and two operators $\sqcup$ and $\sqcap$ that respectively computes the least upper bound and the greatest lower bound of two abstract values. The domain $\mathcal{A}$ is also chosen to have efficient algorithms for $\sqcup$ and $\sqcap$ and a decision procedure for $\sqsubseteq$. An *abstract program property* is represented as a tuple $\boldsymbol{a} \stackrel{def}{=} (a_{\mathsf{entry}}, \ldots, a_{\mathsf{exit}})$ that associates an abstract value to each program node.

In order to conduct proofs based on the Floyd-Hoare method in a deductive system for first order logic (FOL), an abstract value (which is a assertion associated to a program node) must be expressible in first order logic. A large class of abstract domains hold this condition, as *polyhedras, octogons, intervals, difference bound matrices, quantified domains,...*

Before presenting the instrumentation of the static analyzer ENKIDU, we specify the general architecture of such a tool. A static analyzer computes an abstract program property that is the fixpoint of the abstract transfer function associated to the program which merges the effect of each transition of the program. Formally, the abstract semantics of a program P is the least solution $\boldsymbol{a}$ of the equation

[1] The COQ environment has recently been accepted by governmental authorities in a certification at the highest level of assurance of the Common Criteria for Security [8].

(†) $\boldsymbol{a} = \boldsymbol{a}^0 \bigsqcup_{\tau_{qq'} \in P} f_{\tau_{qq'}}(\boldsymbol{a})$ where $\boldsymbol{a}^0$ denotes the initial conditions of the program, and $f_{\tau_{qq'}}$ is the transfer function associated to the transition $\tau_{qq'}$. $f_{\tau_{qq'}}$ computes the abstract value associated to location q' from the abstract value at location q and lets the other values unchanged (it can be seen as a predicate transformer that captures the effect of transition $\tau_{qq'}$).

We now give a typical implementation of an analyzer that computes a solution of Equation (†) in an iterative way. Starting from the initial program property $\boldsymbol{a}^0$ the analyzer builds a more precise property by computing the sequence $(\boldsymbol{a}^i)_{i\in\mathbb{N}}$ defined by

$$\begin{cases} \boldsymbol{a}^i & = \mathsf{SearchFixPoint}([\boldsymbol{a}^0, \ldots, \boldsymbol{a}^{i-1}]) \\ \boldsymbol{a}^{i+1} & = \mathsf{Step}(P, \boldsymbol{a}^i) \stackrel{def}{=} \boldsymbol{a}^i \bigsqcup_{\tau_{qq'} \in P} f_{\tau_{qq'}}(\boldsymbol{a}^i) \end{cases}$$

until it reaches stability, that is, a rank n such that the predicate

$$\mathsf{Stability}(\boldsymbol{a}^{n+1}, \boldsymbol{a}^n) \stackrel{def}{=} \bigwedge_{q \in 1..|\boldsymbol{a}|} \boldsymbol{a}_q^{n+1} \sqsubseteq \boldsymbol{a}_q^n$$

returns $true$ meaning that $\boldsymbol{a}^n$ is a fixpoint of Equation (†) (where a_q^n is the assertion from *abstract program property* $\boldsymbol{a}^n$ at node q).

Usually, the least solution of the equation is not computable and analyzers uses *widening* operators to reach a post-fixpoint of Equation (†). This feature of the analyzers is part of the function SearchFixPoint, that controls termination but it is not relevant for correctness. Therefore, only the Stability predicate and the Step function must be instrumented to obtain certificates. In order to obtain inductive invariants, it is sufficient to reach a rank n such that $\boldsymbol{a}^{n+1} \sqsubseteq \boldsymbol{a}^n$, meaning that the property $\boldsymbol{a}^n$ is preserved by the transitions. This abstract inclusion is guaranteed when the analyzer stops, no matter if it reaches a fixpoint or a post-fixpoint [10].

3 A generic instrumentation of a static analyzer

For the sake of clarity, the certifying functions are given in a pseudo-programming language which we expect to be self-explanatory.

3.1 Instrumentation of the fixpoint computation

Imagine that the analyzer is called on the program P with $\boldsymbol{a}^0$ as initial condition. It computes the sequence $(\boldsymbol{a}^i)_{i\in\mathbb{N}}$ of properties until a fixpoint, say $\boldsymbol{a}^n$, is reached and returns it as an inductive program property. The certifying analyzer, $\mathsf{Analyzer}^\pi$ runs the uncertifying one to get $\boldsymbol{a}^n$. Next, it replays only the last iteration that led to stability using the certifying version of the functions Step and Stability (denoted by Step^π and $\mathsf{Stability}^\pi$). Hence, the overhead of proof-generation is very limited. The extra computations provide the evidences needed for building the certificate.

$$\begin{array}{l} \mathsf{Analyzer}^\pi(P, \boldsymbol{a}^0) \stackrel{def}{=} \mathsf{let}\ \boldsymbol{a}^n = \mathsf{Analyzer}(P, \boldsymbol{a}^0) \\ \quad \mathsf{and}\ (\boldsymbol{a}^{n+1}, \pi_1) = \mathsf{Step}^\pi(P, \boldsymbol{a}^n) \\ \quad \mathsf{and}\ (\_, \pi_2) = \mathsf{Stability}^\pi(\boldsymbol{a}^{n+1}, \boldsymbol{a}^n) \\ \quad \mathsf{and}\ \pi = \dfrac{\pi_1 \quad \pi_2}{\{\boldsymbol{a}^n\}\ P\ \{\boldsymbol{a}^n\}}\ {}^{consequence} \quad \mathsf{in}\ (\boldsymbol{a}^n, \pi) \end{array}$$

Let us temporarily assume that Step^π and $\mathsf{Stability}^\pi$ are available and instrumented to generate the proofs that their results satisfy their correctness property. Here is the goal of the extra computations: $\mathsf{Step}^\pi(P, \boldsymbol{a}^n)$ computes $\boldsymbol{a}^{n+1}$ and provides the proof π_1 of the Hoare triple $\{\boldsymbol{a}_n\}\ P\ \{\boldsymbol{a}_{n+1}\}$. Of course, $\mathsf{Stability}^\pi$ called with $(\boldsymbol{a}^{n+1}, \boldsymbol{a}^n)$ returns $true$ since stability was reached with $\boldsymbol{a}_n$ and it provides a proof π_2 of the implication $\boldsymbol{a}^{n+1} \Rightarrow \boldsymbol{a}^n$. These proofs are then combined using the *consequence* (which extends the transitivity of the implication to Hoare triple) into the final certificate:

$$\mathsf{Stability}^{\pi}(\boldsymbol{a},\boldsymbol{a'}) \stackrel{def}{=} \text{let } (b_q,\pi_q) = a_q \sqsubseteq^{\pi} a'_q \text{ for each } q \in 1..|\boldsymbol{a}| \text{ in } (\bigwedge_{q\in 1..|\boldsymbol{a}|} b_q, \pi)$$

$$\text{where } \pi = \left\{ \dfrac{\dfrac{\left.\dfrac{\pi_q}{a_q \Rightarrow a'_q} \quad \cdots \right\} \text{for each } q \in 1..|\boldsymbol{a}|}{\bigwedge_{q\in 1..|\boldsymbol{a}|} a_q \Rightarrow a'_q}\ \wedge_{intro}}{\boldsymbol{a} \Rightarrow \boldsymbol{a'}}\ tuple \right.$$

$$\mathsf{Step}^{\pi}(P,\boldsymbol{a}) \stackrel{def}{=} \text{let } (a'_{q'},\pi_\tau) = f^{\pi}_{\tau}(a_q) \text{ for each } q \xrightarrow{\tau} q' \in P \text{ and } \boldsymbol{a'} = \boldsymbol{a} \bigsqcup_{q \xrightarrow{\tau} q' \in P} a'_{q'} \text{ in } (\boldsymbol{a'},\pi)$$

$$\text{where } \pi = \left\{ \dfrac{\dfrac{\left.\dfrac{\pi_\tau}{a_q \Rightarrow wp(\tau, a'_{q'})}\ sem_2 \ \cdots \right\} \text{for each } q \xrightarrow{\tau} q' \in P}{\bigwedge_{q \xrightarrow{\tau} q' \in P} \{a_q\}\ \tau\ \{a'_{q'}\}}\ \wedge_{intro}}{\{\boldsymbol{a}\}\ P\ \{\boldsymbol{a'}\}}\ sem1 \right.$$

Figure 2: The certifying versions of Step and Stability

$$\dfrac{\overbrace{\{\boldsymbol{a}^n\}P\{\boldsymbol{a}^{n+1}\}}^{\pi_1} \quad \overbrace{\boldsymbol{a}^{n+1} \Rightarrow \boldsymbol{a}^n}^{\pi_2}}{\{\boldsymbol{a}^n\}\ P\ \{\boldsymbol{a}^n\}}\ consequence$$

The certifying functions Step^{π} and $\mathsf{Stability}^{\pi}$ are presented in Figure 2. Again, the instrumentation consists in short proof patterns that simply combine subproofs provided by more fundamental certifying functions: f^{π} and $\sqsubseteq^{\pi}$. The two algorithms consist in collecting the results and the proofs returned by distinct computations (the application of f^{π} to all transitions of the program for Step^{π}, and the evaluation of $\sqsubseteq^{\pi}$ component by component for $\mathsf{Stability}^{\pi}$) and then, to compute the main result and to separately combine the subproofs using deduction and semantics rules.

The proofs are provided in an intermediate format that is then automatically translated into COQ syntax. A proof is a deduction tree represented by an OCAML data structure. It consists of a combination of deduction rules which capture logical deduction, semantic reasoning, rewriting and provide the ability to use mathematical axioms or lemmata as long as they are available in the target proof-checker. Logical connectors and quantifiers are handled by the rules of the natural deduction. The semantics steps are performed using the following deduction rules:

$$\dfrac{\bigwedge_{\tau_{qq'}\in P} \{a_q\}\ \tau_{qq'}\ \{a_{q'}\}}{\{\boldsymbol{a}\}\ P\ \{\boldsymbol{a}\}}\ (sem_1) \qquad \dfrac{a \Rightarrow wp(\tau, a')}{\{a\}\ \tau\ \{a'\}}\ (sem_2)$$

$$\textit{with } wp(g,a) \stackrel{def}{=} g \Rightarrow a \textit{ and } wp(v := e, a) \stackrel{def}{=} a[v/e]$$

The rule (sem_1) is used to split the proof of the global abstract invariant $\boldsymbol{a} \stackrel{def}{=} (a_{\mathsf{entry}}, a_q, a_{q'} \ldots, a_{\mathsf{exit}})$ into the proofs of one Hoare triple for each transition of the program. Rule (sem_2) relates the validity of a triple to Dijkstra's calculus of the *weakest precondition guaranteeing a property* after execution of a transition, denoted by wp [12].

The proof pattern of $\mathsf{Stability}^{\pi}$ is generic, in the sense that it will work for any analyzer if the abstract comparison operator is instrumented. Indeed, the proof only relies on the correctness property of $\sqsubseteq$: It exactly uses the fact $a \Rightarrow a'$ that must hold if $a \sqsubseteq a'$ [10], since $a, a' \in$ FOL. Concerning the certifying version of $\sqsubseteq$ we cannot reason any further on a generic analyzer. Each analyzer is based on an abstract domain with a specific comparison operator. We give an example of the instrumentation of $\sqsubseteq$ in section 4 in the particular case of ENKIDU, an analyzer for array-processing programs.

We now come to the generation of the proof (π_τ in Figure 2) of the Hoare triple $\{a_q\}\ \tau\ \{a'_{q'}\}$ for each transition τ of the program. In previous works [7, 27], this was done by instrumenting the abstract

transfer functions of the static analyzers. But, the transfer function represents a great amount of code that is specific to each static analyzer. It is usually implemented as a large case analysis – the heuristics – to choose the treatment that gives the most precise result. Avoiding the instrumentation of each of these special treatments is worth the game. In this section we explain how the proof generation can be done without modifying the transfer function f_τ ; instead, we instrument some operators of the abstract domain. The effort of instrumentation is amortized since the abstract domains are usually defined as libraries shared by many analyzers.

3.2 Instrumentation of the transfer function

The key idea to obtain an instrumented version f_τ^π of the transfer function f_τ is to insert the assertion $a_q \sqsubseteq_{\mathcal{A}}^{\pi} wp(\tau, a'_{q'})$ after the computation $a'_{q'} = f_\tau(a_q)$. If the analysis is correct, the verification of the assertion must succeed and produce the proof π_τ of the implication $a_q \Rightarrow wp(\tau, a'_{q'})$ required by the function Step^π of Figure 2. This way we avoid the study of the code of f_τ. Instead we provide in Figure 3 a generic instrumentation of the transfer function that reuses f_τ and then selects among three proof-patterns depending on the type of the transition $q \xrightarrow{\tau} q'$ (a guard or an assignment) and the form of $wp(\tau, a'_{q'})$. The rest of this section focuses on this instrumentation, it has been designed to ensure the completeness of the proof-generation in the following sense: When the analyzer discovers a valid property, we are guaranteed that its instrumented version will generate a valid proof that the program satisfies the property.

3.2.1 Guarded transitions.

The weakest precondition calculus for a guarded transition $q \xrightarrow{g} q'$ rewrites the proof obligation $a_q \Rightarrow wp(\tau, a'_{q'})$ into $a_q \Rightarrow (g \Rightarrow a'_{q'})$ which is equivalent to $a_q \wedge g \Rightarrow a'_{q'}$. We can take advantage of this reformulation since the abstract domain is closed under conjunction. Indeed, the proof of the implication is mainly built on the sub-proof π_1 provided by $a_q \wedge \mathcal{N}(g) \sqsubseteq_{\mathcal{A}}^{\pi} a'_{q'}$ where $\mathcal{N}$ denotes the normalization function associated to the abstract domain that transforms guards of the program into their abs-tact representations. The normalization function must be instrumented to establish (sub-proof π_2) that $\mathcal{N}(g)$ is a safe approximation of g, that is $g \Rightarrow \mathcal{N}(g)$. The proof-pattern of Figure 3 for guarded transition is a combination of the sub-proofs π_1 and π_2.

3.2.2 Assignment transitions.

We distinguish two cases according to whether the weakest precondition $wp(\tau, a'_{q'})$ belongs to the abstract domain $\mathcal{A}$ or not.

1a. The propitious case: When $wp(a'_{q'}, \tau) \in \mathcal{A}$, the premise and the conclusion of the implication belong to abstract domain so the proof of the implication is provided by the computation $a_q \sqsubseteq_{\mathcal{A}}^{\pi} wp(a'_{q'}, \tau)$ with the instrumented operator $\sqsubseteq_{\mathcal{A}}^{\pi}$.

1b. The common case: Most of the time $wp(a'_{q'}, \tau) \notin \mathcal{A}$ – meaning that the formula produced by wp is not in the form required by the abstract domain – but it can be expressed in $\mathcal{A}$ by means of a normalisation step. This can be illustrated on the domain of interval constraints: the weakest precondition of $x \in [min_x, max_x]$ for the statement `x:=x+k` (where `k` is an integer constant) replaces x by $x + k$, that leads to a constraint $x+k \in [min_x, max_x]$ bearing on an expression, not on a variable. However the normalization is straightforward that leads to $x \in [min_x - k, max_x - k]$ which belongs to the domain of intervals. Actually, we are brought back to the previous case by systematically applying the normalization function $\mathcal{N}$ to $wp(a'_{q'}, \tau)$. Note that when the normalization produces a equivalent form of a formula ϕ, its instrumented version $\mathcal{N}^\pi$ outputs a proof of the implication $\mathcal{N}(\phi) \Rightarrow \phi$ (accuracy) in addition to that of the implication $\phi \Rightarrow \mathcal{N}(\phi)$ (safe approximation).

2. The worst case appears when $wp(a'_{q'}, \tau) \notin \mathcal{A}$ and it cannot be expressed in the abstract domain. For instance, the effect of the assignment `x:=y+z` on the constraint $y \in [min_y, max_y] \wedge z \in [min_z, max_z]$ is captured by the post-condition $x \in [min_y + min_z, max_y + max_z]$. In the other direction, wp leads to $y + z \in [min_x, max_x]$ where min_x and max_x are values and this cannot

$$f^{\pi}_{\tau}(a_q) \stackrel{def}{=} \text{let } a'_{q'} = f_\tau(a_q) \text{ in}$$
$$\text{match } \tau \text{ with}$$
$$\mid (q \xrightarrow{guard\ g} q') \rightarrow \text{let } (\_, \pi_1) = a_q \wedge \mathcal{N}(g) \sqsubseteq^{\pi}_{\mathcal{A}} a'_{q'} \text{ and } (\_, \pi_2, \_) = \mathcal{N}^{\pi}(g) \text{ in } (a'_{q'}, \pi_\tau)$$

$$\text{where } \pi_\tau = \left\{ \dfrac{\dfrac{\overbrace{a_q}^{H_1} \quad \dfrac{\overbrace{g}^{H_2} \quad \pi_2}{g \Rightarrow \mathcal{N}(g)} \Rightarrow_{elim}}{a_q \wedge \mathcal{N}(g)} \wedge_{intro} \quad \dfrac{\pi_1}{a_q \wedge \mathcal{N}(g) \Rightarrow a'_{q'}} \Rightarrow_{elim}}{\dfrac{a'_{q'}}{a_q \Rightarrow (g \Rightarrow a'_{q'})} \Rightarrow_{intro\ H_1}, \Rightarrow_{intro} H_2}\ {\scriptstyle def.\ wp} \right.$$

$$\mid (q \xrightarrow{t:=e} q') \rightarrow \text{ try let } (\tilde{a}, \_, \pi_1) = \mathcal{N}^{\pi}(wp(\tau, a_q)) \text{ and } (\_, \pi_2) = a_q \sqsubseteq^{\pi}_{\mathcal{A}} \tilde{a} \text{ in } (a'_{q'}, \pi_\tau)$$

$$\text{where } \pi_\tau = \left\{ \dfrac{\dfrac{\pi_2}{a_q \Rightarrow \mathcal{N}(wp(\tau, a'_{q'}))} \quad \dfrac{\pi_1}{\mathcal{N}(wp(\tau, a'_{q'})) \Rightarrow wp(\tau, a'_{q'})}}{a_q \Rightarrow wp(\tau, a'_{q'})} \right.$$

$$\text{with Fail}(\mathcal{N}^{\pi}) \rightarrow \text{let } (a'_{q'[t/e]}, \pi_1) = eval^{\pi}(e, a_q) \text{ in } (a'_{q'}, \pi_\tau)$$

$$\text{where } \pi_\tau = \left\{ \dfrac{\dfrac{\pi_1}{a_q \Rightarrow a'_{q'[t/e]}}}{a_q \Rightarrow wp(t := e, a'_{q'})}\ {\scriptstyle def.\ wp} \right.$$

Figure 3: The generic instrumentation of a transfer function f_τ

be in the form of conjunction of interval constraints, unless we knew how to split back min_x and max_x. Therefore, the $\sqsubseteq^{\pi}_{\mathcal{A}}$ operator cannot be used to prove the implication $y \in [min_y, max_y] \wedge z \in [min_z, max_z] \Rightarrow y + z \in [min_x, max_x]$. To solve that case we need to look back at the general treatment of an assignment, say `x:=y+z`, in abstract interpretation: The assigned variable is set to the abstract value of the right-hand side expression which is evaluated in the abstract context a_q, that is, using the current abstract values of y, z and the abstract operator $\widehat{+}$ defined as $[min_y, max_y] \widehat{+} [min_z + max_z] \stackrel{def}{=} [min_y + min_z, max_y + max_z]$. The missing justification is obtained by instrumenting the $\widehat{+}$ operator to produce a proof of its correctness property which is exactly the needed implication: $y \in [min_y, max_y] \wedge z \in [min_z, max_z] \Rightarrow y + z \in [min_y + min_z, max_y + max_z]$.

The completeness of the proof generation is guaranteed at the price of the instrumentation of the normalization function, the $\sqsubseteq_{\mathcal{A}}$ operator and the arithmetic operators of the abstract domain. Compared with previous work [7, 27] our instrumentation is (1) lighter since it avoids the instrumentation of the most complex and specific part of an analyzer – its transfer function and (2) it is rapidly amortized since the instrumentation concerns the (reusable) abstract domain and is not specific to the analyzer.

Finally, the trusted computing base (TCB) of our certifying analyzer consists in the COQ proof-checker and the definition of the semantics of guards and assignments as a weakest precondition calculus in COQ. Note that the proof patterns used in the instrumentation do not have to be trusted: If they contain logical bugs then the certifying static analyzer will produce non-valid deduction trees which will be rejected by the proof-checker.

4 Application to an existing static analyzer

We applied our instrumentation principle to an existing static analyzer, ENKIDU [16], which discovers properties of array-processing programs using an abstract interpretation based on a lattice defined on a quantified logical domain [15]. ENKIDU determines a symbolic division $\mathscr{S}$ (a partition) of the array indexes into a finite number of slices $\varphi_1, \ldots, \varphi_N$ and then uses this fixed division as a basis for an abstract interpretation of the program. The main abstract domain $\mathcal{A}$ used in ENKIDU is the class of first order logic

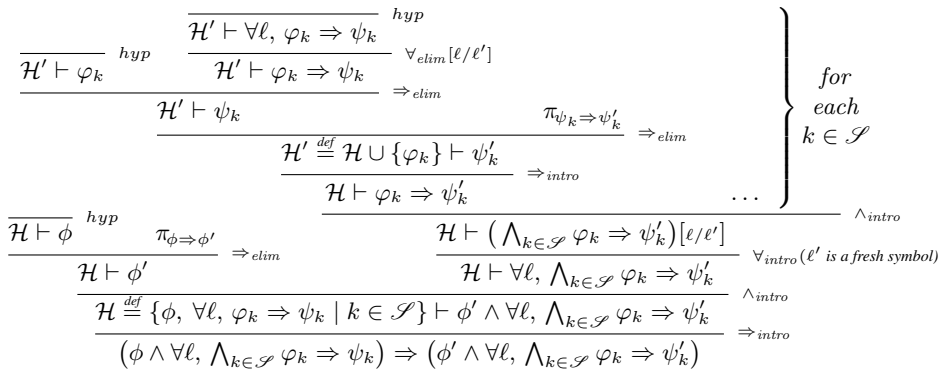

Figure 4: The proof scheme for the certifying version of $\sqsubseteq_{\mathcal{A}}$

array content properties of the form

$$\phi \wedge \forall \ell, \bigwedge_{k \in \mathscr{S}} \varphi_k \Rightarrow \psi_k$$

where the constraints ϕ, φ_k and ψ_k are elements of the *zone abstract domain* by Miné [21], that are conjunctions of constraints of the form $x - y \leq k$ where x, y are integer variables and k is a constant, implemented as *difference bound matrices* (DBM for short). Formula ϕ captures unquantified constraints on variables of the program *e.g.* $j > 1$, φ_k uses the logical variable ℓ to define a slice of the array *e.g.* $\ell \leq j - 1$, and ψ_k is a property on array cells *e.g.* $A[\ell] \leq A[\ell + 1]$. The array content property, $j > 1 \wedge \forall \ell, (\ell \leq j - 1 \Rightarrow A[\ell] \leq A[\ell + 1])$, means that the array A is sorted until index j.

4.1 The instrumentation of ENKIDU

Satisfiability/unsatisfiability of a DBM is decidable as well as the implication between two DBM, using a normalization function based on the *Floyd-Warshall algorithm*. The implication criteria is used to compare the strength of DBM constraints and provides an ordering on DBM. Therefore, the DBM (resp. the array properties) forms a lattice of formulæ ordered by $\sqsubseteq_D$ (resp. $\sqsubseteq_{\mathcal{A}}$) that corresponds to the logical implication on this subset of FOL. However we keep the symbols $\sqsubseteq_D$ and $\sqsubseteq_{\mathcal{A}}$ to stress that each one corresponds to a specialized strategy for deciding implication on a specific class of properties (DBM and array properties).

The lattice of array properties $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}, \sqcup_{\mathcal{A}}, \sqcap_{\mathcal{A}}, \top_{\mathcal{A}}, \bot_{\mathcal{A}})$ is built on top of the DBM lattice. The operators $\sqcup_{\mathcal{A}}$ and $\sqcap_{\mathcal{A}}$ which are intensively used during the fixpoint computation are meaningless for justification ; only the ordering operator $\sqsubseteq_{\mathcal{A}}$ matters. It is used to detect stability of the program property (see section 3.1), that is $a_q^{n+1} \sqsubseteq_{\mathcal{A}} a_q^n$ for each node q of the program, and must produce the proof of $a_q^{n+1} \Rightarrow a_q^n$. The predicate $\sqsubseteq_{\mathcal{A}}$ on array properties is implemented using the comparison operator $\sqsubseteq_D$ on DBM: For a fixed partition $\mathscr{S}$, $\left(\phi \wedge \forall \ell, \bigwedge_{\mathscr{S}} \varphi_k \Rightarrow \psi_k\right) \sqsubseteq^{\pi}_{\mathcal{A}} \left(\phi' \wedge \forall \ell, \bigwedge_{\mathscr{S}} \varphi_k \Rightarrow \psi'_k\right)$ holds if the following conditions on DBM are satisfied: $\phi \sqsubseteq_D \phi'$ and $\bigwedge_{k \in \mathscr{S}} \psi_k \sqsubseteq_D \psi'_k$. The φ_k's can be ignored as the partition is fixed for all array properties during an execution of ENKIDU. As an example of justification, the certifying version of $\sqsubseteq_{\mathcal{A}}$ is presented in Figure 4 where

- the proof $\pi_{\phi \Rightarrow \phi'}$ comes from the evaluation of $\phi \sqsubseteq^{\pi}_D \phi'$
- all the proofs $\pi_{\psi_k \Rightarrow \psi'_k}$ are collected during the evaluation of $\bigwedge_{k \in \mathscr{S}} \psi_k \sqsubseteq^{\pi}_D \psi'_k$.

The proof pattern can seem technical but it was not difficult to find: it is almost directed by the structure of the formula. In the current version, the instrumentation of $\sqsubseteq_D$ is just a call to the `omega` tactic that decides Presburger arithmetic. Specific and optimized proof patterns for $\sqsubseteq_D$ are available in [14] (page 103) but have not yet been implemented. We guess that they could significantly speed up certificate checking in COQ since all the time spent by `omega` would be transferred at certificate generation and would be done by specialized algorithm developed in a more efficient language.

4.2 Dealing with array aliases

In order to prove properties of array-processing program in a proof system based on the calculus of the weakest precondition, one must take care of the aliases between indexes of array cells. Indeed, Hoare assignment rule $\{\Phi[_{t/e}]\}$ `t:=e` $\{\Phi\}$ is unsound for array assignments. Indeed, consider an assertion about the cell $A[l]$, the assignment $A[\mathtt{e}] := \mathtt{r}$ must be treated carefully due to potential aliasing between $A[\mathtt{e}]$ and $A[l]$ when $l = \mathtt{e}$. For example, it does not handle aliases between indexes of arrays. Indeed, it can demonstrate the incorrect triple

$$\{\}\ \mathtt{A[i]:=0;\ j:=i;\ A[j]:=1}\ \{A[i] = 0\}$$

and fails to prove the following valid triple, see Figure 5 for details

$$\{\}\ \mathtt{i:=j;\ T[j]:=1}\ \{T[i] = 1\}.$$

Morris [22] shows that the alias problem can be solved if all the potential aliases are made explicit in the assertion. It provides a new set of rules that can correctly deal with aliases for non nested array references, meaning that expressions like $A[A[i]]$ are forbidden. A sound and complete solution has been proposed by Hoare and Wirth [18] and McCarthy and Painter [20], that is implemented in the CADUCEUS tool [13] of the WHY plateform for proving properties of imperative programs in COQ using a *wp* calculus. Consider the assignment $A[\mathtt{e}] := \mathtt{r}$ and a property $\Phi(A[l])$. The idea is to represent the array A as a function[2], initially denoted by the λ-term $(\lambda i_0.A[i_0])$. Then, instead of modifying the cell $A[\mathtt{e}]$ through the substitution $[_{A[\mathtt{e}]/\mathtt{r}}]$, the array itself is modified. Its definition as a function is updated into $(\lambda i_1.$ if $(i_1 = \mathtt{e})$ then $\mathtt{r}$ else $((\lambda i_0.A_0[i_0])i_1)$, which reduces to $(\lambda i_1.$ if $(i_1 = \mathtt{e})$ then $\mathtt{r}$ else $A[i_1])$. Then, the value of $A[l]$ in the property $\Phi(A[l])$ is the result of the function applied to l that is $\Phi($if $(l = \mathtt{e})$ then $\mathtt{r}$ else $A[l])$. The alias problem is solved by explicitly testing the equality between i_1 and the modified cell at index $\mathtt{e}$. Computing the abstraction of $\Phi(A[l])$ mean processing the β-reductions and then using a case analysis to change the previous expression into an equivalent statement in FOL $(l = \mathtt{e}) \Rightarrow \Phi(\mathtt{r}) \bigwedge \neg(l = \mathtt{e}) \Rightarrow \Phi(A[l])$ before applying the abstraction function α. We implement this solution in our framework allowing the justify version of ENKIDU to certify array-processing programs.

4.3 Implementation and experimentation

At the starting point of this work we have a prototype implementation of the ENKIDU analyzer. The code of the analyzer consists of about 200 functions (7000 lines of Ocaml) ; five of those were instrumented: Analyzer, Step, Stability, and the comparison operators $\sqsubseteq_{\mathcal{A}}$ and $\sqsubseteq_{D}$ of the two lattices (of array properties and DBM) used in the abstract interpretation. The current version of the certifying analyzer is also at prototype stage, totally unoptimized and produces certificates which combine tactics and proof terms. The instrumentation resulted in less than 1200 additional lines which mainly corresponds to the proof patterns. We still need some experimentation to find the good trade-off between proof size and checking time. Above all, it depends on the use of certificate: in a certification process (of an application for smart cards for instance [8]), the certificate is checked once and not sent over the Internet; whereas in a PCC architecture both time and size matter and the checker must be provided only with the evidences which cannot be discovered by an automatic theorem prover. In the context of PCC, it would probably be preferable to provide the user with the program, the property and the justified analyzer, instead of the proof since that one can be quickly regenerated by the justified analyzer from the already discovered property. This experiment convince us of the feasibility of instrumenting an existing validation tool (even at prototype stage) so that it automatically builds certificates of its verdicts in the form of foundational proofs. The most important for us was to be able to generate machine-checkable foundational proofs on non-trivial program properties. To achieve this as soon as possible we used tactics of the COQ system every time it was possible. Using COQ tactics drastically reduces proof size at the price of an increase of proof-checking time. Indeed, a tactic is the name of a proof-search algorithm that is run by the proof-assistant, producing a proof term which is then checked by the proof-checker.

[2]The same idea can be rephrased using the $store/select$ axioms for arrays.

Hoare's rule for assignment $\{\phi[v/e]\}$ `v:=e` $\{\phi\}$ is not sound in the presence of alias. Indeed, it can demonstrate the incorrect triple $\{\}$ `A[i]:=0; j:=i; A[j]:=1` $\{A[i] = 0\}$

$$\begin{array}{lll}
 & & \{(A[i] = 0)[_{A[j]/1}]\} \quad \texttt{A[j]:=1} \quad \{A[i] = 0\} \\
 & \{(A[i] = 0)[_{j/i}]\} \quad \texttt{j:=i} \quad \{A[i] = 0\} & \\
\{(A[i] = 0)[_{A[i]/0}]\} \quad \texttt{A[i]:=0} \quad \{A[i] = 0\} & & \\
\{0 = 0\} \equiv \{\} & &
\end{array}$$

However Hoare's rule is sound for formulæ with explicit alias. We then use the sound assignment rule $\{Æ_v(\phi)[_{v/e}]\}$ `v:=e` $\{\phi\}$ where the function $Æ$ transforms a formula into an equivalent formula with explicit alias before we apply Hoare's substitution.

$$\begin{array}{lll}
 & & \{\big(Æ_{A[j]}(A[i] = 0)\big)[_{A[j]/1}]\} \ \texttt{A[j]:=1} \quad \{A[i] = 0\} \\
 & & \{(i = j \Rightarrow A[j] = 0 \wedge i \neq j \Rightarrow A[i] = 0)[_{A[j]/1}]\} \\
 & & \{(i = j \Rightarrow 1 = 0) \wedge (i \neq j \Rightarrow A[i] = 0)\} \equiv_{(1)} \\
 & & \{(i \neq j) \wedge (i \neq j \Rightarrow A[i] = 0)\} \equiv_{(2)} \\
\{(i \neq j \wedge \ldots)[_{i/j}]\} \quad \texttt{j:=i} \quad \{i \neq j \wedge A[i] = 0\} & & \\
\{i \neq i \wedge \ldots\} \equiv_{(3)} \{\mathit{false}\} & &
\end{array}$$

Hoare's assignment rule is incomplete in presence of alias. For instance, it fails to prove the triple $\{\}$ `i:=j; T[j]:=1` $\{T[i] = 1\}$

$$\begin{array}{lll}
 & & \{(T[i] = 1)[_{T[j]/1}]\} \quad \texttt{T[j]:=1} \quad \{T[i] = 1\} \\
\{(T[i] = 1)[_{i/j}]\} \quad \texttt{i:=j} \quad \{T[i] = 1\} & & \\
\{T[j] = 1\} \not\equiv \{\} & &
\end{array}$$

However the assignment rule is complete for formulæ with explicit alias.

$$\begin{array}{lll}
 & & \{Æ_{T[j]}(T[i] = 1)[_{T[j]/1}]\} \quad \texttt{T[j]:=1} \quad \{T[i] = 1\} \\
 & \{(i = j \Rightarrow T[j] = 1 \bigwedge i \neq j \Rightarrow T[i] = 1)[_{T[j]/1}]\} & \\
 & \{i = j \Rightarrow 1 = 1 \bigwedge i \neq j \Rightarrow T[i] = 1\} \equiv_{(4)} & \\
 & \{i = j \bigwedge i \neq j \Rightarrow T[i] = 1\} \equiv_{(5)} & \\
\{(i = j)[_{i/j}]\} \quad \texttt{i:=j} \quad \{i = j\} & & \\
\{j = j\} \equiv_{(6)} \{\} & &
\end{array}$$

Formulæ with explicit alias are automatically simplified by the means of propositional simplifications such as: (1) $A \Rightarrow \mathit{false} \equiv \neg A$ and (2) $A \wedge (A \Rightarrow B) \equiv A \wedge (\neg A \vee B) \equiv (A \wedge \neg A) \vee (A \wedge B) \equiv A \wedge B$ and (3) $\mathit{false} \wedge A \equiv \mathit{false}$ and (4) $A \Rightarrow \mathit{true} \equiv A$ and (5) $A \wedge (\neg A \Rightarrow B) \equiv A \wedge (A \vee B) \equiv A$ and (6) $x = x \equiv \mathit{true}$.

Figure 5: The alias problem with Hoare's rule for assignment

It is worth noticing that the dependencies between ENKIDU and its instrumented version are very limited since the instrumentation almost consists in the creation of certifying functions that call the original ones. Hence, the instrumentation is not sensitive to minor modifications of the analyzer. The justifications of the verdicts required some extra developments for the definition of proof patterns, the weakest precondition calculus, and the translation of proofs into the COQ syntax. These developments are independent of ENKIDU and can be reused for the instrumentation of other tools.

5 Conclusion and future work

There are several ways to increase the level of confidence we can place in a validation tool. In increasing level of trustability, one can (1) prove the correction of the tool principle by pencil and paper, or (2) establish the correction of the tool algorithm in a prover, or (3) generate a certificate for each particular run of the tool. The last degree, (4) formally proving the correctness of the *implementation*, does not seem achievable.

In our opinion, justification of validation tools by instrumentation provides the following advantages: *Accuracy*, it provides a proof of the actual implementation with all its undocumented extensions and optimizations. *Easiness*, the algorithm and its implementation need not to be proved for every possible entries, but only on distinct runs. *Robustness*, if the implementation of the tool changes slightly there is often no need to adapt certificate generation. *Privacy*, there is no need to give access to the tool and its algorithms to guarantee its verdict.

In this report, we explained how a static analyzer can be instrumented to automatically produce certificates of its verdicts in the form of foundational proofs. The principle is illustrated on ENKIDU which discovers properties of array-processing programs [16]. By instrumenting only the comparison operators of the abstract domains and combining their proofs in the three main functions (Analyzer, Step and Stability) the analyzer is able to produce a COQ proof of the inductiveness of the program properties using the Floyd-Hoare proof technique. The generated proofs are a large combination of simple steps produced by the instrumented functions during the last iteration of the fixpoint computations.

We did not re-implement the analysis in a prover as tactics but rather we followed the tool computations as a proof strategy in order to take advantage of the know-how of the developer. In contrast to previous work on certification of static analyzers, we achieve three challenges: (1) We produce foundational proofs (each minor step must be explicitly justified) for a trusted proof-checker – this provides a very high level of confidence in the certificate; (2) Using proof patterns we can guarantee completeness of the proof generation in the following sense: if the property discovered by the analyzer is valid then the instrumentation returns a valid proof assessing the property; (3) We reduce and factored out the specific instrumentation of the analyzer to the comparison operators of its abstract domains, especially we avoid the complex instrumentation of the abstract transfer function.

As mentioned before, the use of tactics reduces the proof size but decreases the proof-checking performance by postponing the proof-search to the verification phase. Future work will focus on reconciling the two contradictory goals: reducing the size of proofs and increasing efficiency of the proof-checking. This can be achieved following [5, 9] by designing specialized proof-checkers themselves certified in COQ. Then, applying our instrumentation methodology to a more sophisticated tool (*e.g.* the industrial C programs analyzer ASTRÉE [11]) should conclude the feasibility of this certification approach.

References

[1] A. Appel, N. Michael, A. Stump, and R. Virga. A Trustworthy Proof Checker. *Journal of Automated Reasoning*, 31(3-4):231–260, 2003. 1.1, 1.4

[2] Andrew W. Appel. Foundational Proof-Carrying Code. In *IEEE Symposium on Logic in Computer Science*, pages 247–258, 2001. 1.4

[3] B. Barras. Verification of the Interface of a Small Proof System in Coq. In *Types for Proofs and Programs*, volume 1512 of *LNCS*, pages 28–45, 1996. 1.4

[4] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006. 1.1, 1.4

[5] Jan Olaf Blech and Benjamin Gregoire. Certifying Code Generation with Coq. In *Compiler Optimization Meets Compiler Verification*, Electronic Notes in Theoretical Computer Science, 2008. (COCV'08). 5

[6] Richard Bornat. Proving pointer programs in hoare logic. In *Conference on Mathematics of Program Construction*, pages 102–126, 2000. (MPC'00). 1.2

[7] Amine Chaieb. Proof-producing program analysis. In *International Colloquium on Theoretical Aspects of Computing*, volume 4281 of *LNCS*, pages 287–301, 2006. (ICTAC'06). 1.3, 1.4, 3.1, 3.2.2

[8] Boutheina Chetali and Quang Huy Nguyen. Industrial Use of Formal Methods for a High-Level Security Evaluation. In *Formal Methods in the Development of Computing Systems*, volume 5014 of *LNCS*, pages 198–213, 2008. (FM'08). 1, 4.3

[9] Evelyne Contejean and Pierre Corbineau. Reflecting Proofs in First-Order Logic with Equality. In *Conference on Automated Deduction*, volume 3632 of *LNCS*, pages 7–22, 2005. (CADE'05). 5

[10] P. Cousot and R. Cousat. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*. ACM Press, 1977. 2, 3.1

[11] P. Cousot, R. Cousot, J. Feret, L Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, pages 21–30, 2005. 5

[12] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. 3.1

[13] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003. 1.4, 4.2

[14] M. Garnacho. *Automatic and formal certification of critical systems by instrumentation of abstract interpreters*. PhD thesis, Université de Grenoble, Grenoble, France, aout 2010. 4.1

[15] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles of Programming Languages*, pages 235–246. ACM Press, janvier 2008. 4

[16] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *ACM Conference on Programming Language Design and Implementation*, pages 339–348. ACM Press, 2008. (PLDI'08). 1.2, 1.4, 4, 5

[17] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 526–538, 2002. 1, 1.4

[18] C.A. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2:335–355, 1973. 4.2

[19] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages*, pages 42–54. ACM Press, 2006. 1.1

[20] John McCarthy and James.A Painter. Correctness of a compiler for arithmetic expressions. In *Symposium in Applied Mathematics*, volume 19 of *Mathematical Aspect of Computer Science*, pages 33–41. AMS, 1967. 4.2

[21] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, dmbre 2004. 4

[22] J. M. Morris. A general axiom of assignment. assignment and linked data structures. a proof of the schorr-waite algorithm. In *Theoretical Foundations of Programming Methodology (Lecture Notes of the 1981 International Marktoberdorf Summer School)*, pages 25–51, 1982. 1.2, 4.2

[23] Kedar S. Namjoshi. Certifying model checkers. In *Computer Aided Verification*, volume 2102 of *LNCS*, pages 2–13, 2001. 1, 1.4

[24] George C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119. ACM Press, 1997. 1.4

[25] George C. Necula. Translation validation for an optimizing compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 83–94, 2000. (PLDI'00). 1.3

[26] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Tools and Algorihtms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 151–166, 1998. (TACAS'98). 1.3

[27] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *Asian Programming Languages and Systems Symposium*, volume 2895 of *LNCS*, pages 230–245, 2003. (APLAS'03). 1.3, 1.4, 3.1, 3.2.2

[28] T. Tan and R. Cleaveland. Evidence-based model checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 455–470, 2002. 1.4