UDC 004.421

KAT and PHL in Coq

David Pereira and Nelma Moreira

DCC-FC & LIACC – University of Porto
Rua do Campo Alegre 1021, 4169-007, Porto, Portugal
{dpereira,nam}@ncc.up.pt

Abstract. In this article we describe an implementation of Kleene algebra with tests (KAT) in the Coq theorem prover. KAT is an equational system that has been successfully applied in program verification and, in particular, it subsumes the propositional Hoare logic (PHL). We also present an PHL encoding in KAT, by deriving its deduction rules as theorems of KAT. Some examples of simple program's formal correctness are given. This work is part of a study of the feasibility of using KAT in the automatic production of certificates in the context of (source-level) *Proof-Carrying-Code* (PCC).

Keywords: Kleene algebra with tests, Hoare logics, interactive theorem proving, and formal verification of software.

1. Introduction

Kleene algebra with tests (KAT) is an equational system for program verification, developed by Dexter Kozen [1], which is the combination of Boolean algebra (BA) and Kleene algebra (KA) [2], the algebra of regular expressions. This algebraic structure permits the interaction between Boolean tests and regular expressions in a very compact way. It was shown to be highly expressive, and it was applied to several formal verification tasks involving communication protocols, basic safety analysis, source-to-source program transformation, concurrency control, compiler optimization and data-flow analysis [3-5].

In particular, KAT subsumes the propositional fragment of Hoare logic (PHL) [6-8] which is a formal system for the specification and verification of programs, and that is currently the base of most tools for checking program correctness. Central to this approach of program verification is the notion of (*pre-post*) conditions that must be valid during the program execution. Given an annotated program, a *verification conditions generator* can extract a set of proof obligations that, if valid, will ensure the program correctness. Proof obligations can be proved using an automatic or an interactive theorem prover, producing a *certificate*.

We aim at explore the adequacy of KAT in the production of certificates in the context of source-level *Proof-Carrying Code* [9]. In this paper, as a first step, we present an implementation of KAT and PHL in the Coq interactive theorem prover [10]. There are several reasons for choosing the Coq system as a development platform. It is a consolidated proof system with a solid set of logical theories. It is able to produce proof objects that can be verified *a posteriori*, either by Coq itself or by a compact *proof-checker*. Finally, besides providing a set of Coq scripts to reasoning in KAT our goal is to develop a decision procedure for the equational fragment of KAT as a Coq tactic. Kamal Aboul-Hosn has implemented an interactive theorem prover for KAT [11], but contrary to our approach, it is a dedicated application.

This paper is organized as follows. In Section 2 we review the main concepts behind KAT and PHL. Section 3 describes how we have implemented BA, KA, KAT, and PHL in the Coq theorem prover. In Section 4 we make an overview of Coq and its features that were intensively used in our implementation. In Section 5 we present three examples of simple *while*-programs verification using our implementation. In Section 6 we comment on the current state of our implementation, and relate it with the enforcement of security mechanisms for Embedded Systems' software. Finally, in Section 7 we draw some conclusions and point some directions to future research.

2. KAT and PHL

In this section we describe KA and KAT, and show how the latter can be used to encode the deductive rules of PHL for the verification of *while*-programs.

2.1. KA

Kleene algebra (KA for short) [2] is an algebraic structure that consists of a set K together with two binary operations $+$ (plus) and $\cdot$ (dot), a unary operation * (Kleene's star), two constants 0 and 1, and that satisfy certain properties. The meaning of the operators depends on the underlying model being considered. Examples of such models are regular expressions over a finite automata, relational models and the *min,*+ algebra [12,1].

There are several ways of axiomatizing a KA. Here we follow the work presented by Dexter Kozen in [12]. The relation between the different KA's that model different domains is also an interesting point in the overall context of KA. By studying these differences we may be able to reason about the essential axiomatic properties of KA. The axiomatization we are going to consider has the advantage of being sound over non-standard interpretations, and leads to a complete deductive system for the universal Horn theory of KA. This is an important property because it provides a decidable procedure for reasoning equationally in KA.

Recall that a *monoid* is a triple $(K,\cdot,1)$ such that:

- K is a set,
- $\cdot$ is associative binary operation, and
- $\cdot$ has a two-side identity 1, that is, $\forall x \in K,\ 1\cdot x = x\cdot 1 = x$

A monoid is said to be a *commutative monoid* if the associated binary operation is commutative. An *idempotent semiring* is an algebraic structure $(K,+,\cdot,0,1)$ such that :

- $(K,+,0)$ is a commutative monoid,
- $(K,\cdot,1)$ is a monoid,
- $+$ is idempotent, that is, $\forall x \in K,\ x + x = x$,
- 0 is a two-sided annihilator for $\cdot$, and
- $\cdot$ distributes over $+$ on both sides.

In a idempotent semiring we define a partial order $\leq$ on the elements of K as follows:

$$x \leq y \Leftrightarrow x + y = y \tag{1}$$

As is usual we abbreviate $x{\cdot}y$ to xy and consider $+ < \cdot < *$ as the precedence relation over the operators of a KA, thus avoiding the usage of parenthesis.

A KA is an idempotent semiring with the Kleene star operator $*$

$$(K,+,\cdot,0,1,*) \tag{2}$$

and that verifies the following axioms (considering $\forall x, y, z \in K$):

$$x + (y + z) = (x + y) + z \tag{3}$$
$$x + y = y + x \tag{4}$$
$$x + 0 = x \tag{5}$$
$$x + x = x \tag{6}$$
$$x(yz) = (xy)z \tag{7}$$
$$1x = x \tag{8}$$
$$x1 = x \tag{9}$$
$$x(y + z) = xy + xz \tag{10}$$
$$(x + y)z = xz + yz \tag{11}$$
$$0x = 0 \tag{12}$$
$$x0 = 0 \tag{13}$$
$$1 + xx^* \leq x^* \tag{14}$$
$$1 + x^*x \leq x^* \tag{15}$$
$$z + xy \leq x \rightarrow zy^* \leq x \tag{16}$$
$$z + yx \leq x \rightarrow y^*z \leq x \tag{17}$$

Axioms (3)-(13) exhibit the idempotent semiring properties of the Kleene Algebra, while axioms (14)-(17) describe the properties of *. The following equalities are also true and are frequently used when reasoning in a KA:

$$x \leq y \rightarrow xz \leq yz \tag{18}$$
$$x \leq y \rightarrow zx \leq zy \tag{19}$$

$$x \leq y \rightarrow x + z \leq y + z \tag{20}$$
$$x \leq y \rightarrow z + x \leq z + y \tag{21}$$
$$x \leq y \rightarrow x^* \leq y^* \tag{22}$$
$$1 \leq x^* \tag{23}$$
$$x \leq x^* \tag{24}$$
$$x^*x^* = x^* \tag{25}$$
$$x^{**} = x^* \tag{26}$$
$$yx = xz \rightarrow y^*x = xz^* \tag{27}$$
$$(xy)^*x = x(yx)^* \tag{28}$$
$$(x + y)^* = x^*(yx^*)^* \tag{29}$$

2.2. KAT

A Kleene algebra with tests (KAT for short) is an extension of a KA which has embedded a BA for performing propositional tests at the same level of KA terms. More formally, a KAT is an algebraic structure

$$(K,B,+,;,^*,\bar{\ },0,1)$$

such that:

- $(K,+,;,^*,0,1)$ is a KA,
- $(B,+,;,\bar{\ },0,1)$ is a BA,
- $B \subseteq K$.

Elements of B are called *tests*. The *negation* operator $\bar{\ }$ is applied only to elements of B. The axioms of a BA are the following (considering $\forall\, b, c, d \in B$):

$$bc = cb \tag{29}$$
$$bb = b \tag{30}$$
$$b + 1 = 1 \tag{31}$$
$$\overline{bc} = \overline{b} + \overline{c} \tag{32}$$
$$b\,\overline{b} = 0 \tag{33}$$
$$b + cd = (b + c)(b + d) \tag{34}$$
$$b + \overline{b} = 1 \tag{35}$$
$$\overline{(b+c)} = \overline{b}\,\overline{c} \tag{36}$$
$$b\,\overline{\overline{b}} = b \tag{37}$$

These axioms, joined together with the axioms of KA (3-17), form the equational system of KAT. The syntax abbreviations for KAT terms are the same as the ones of KA. We reserve the variables $x,y,z,p,q,r,...$ for terms of KA and the variables $a,b,c,d,...$ for referring to tests, thus eliminating any redundancy between the KA part and the BA part of any KAT term.

Important Remarks In the context of the semantics and logic of programs, KAT forms an essential component of *Propositional Dynamic Logic* [13] (PDL). PDL is a combination of *propositions* and *programs*, defined by mutual induction. PDL supports an operator denoted by $\psi?$ which acts as a guard, and succeeding if ψ is true in the *state* of the program (also known as *world* in Modal Logic [14]). Otherwise, if ψ evaluates to false in some state, the program fails/aborts. The programs are of a simple imperative language with conditionals and *while* loops. One can encode these *while*-programs in KAT as follows:

$$\begin{aligned} p; q &= pq \\ \textsf{if}\, b\, \textsf{then}\, p\, \textsf{else}\, q &= bp + \bar{b}q \\ \textsf{while}\, b\, \textsf{do}\, p &= (bp)^*\bar{b} \end{aligned}$$

This formulation of *while*-programs is done in KAT in the same way as it is done PDL. In practice, many simple program manipulations don't need the full power of PDL, while KAT is enough.

KAT also has been applied to several verification tasks in the context of communication protocols, basic safety analysis, concurrency control, and local variable scoping and compiler optimizations [4,15,16].

KAT's equational theory is complete over relational models, and for the free language-theoretical models consisting of regular sets of *guarded strings* [17,18], in the same way as regular sets serve as model to the equational theory of KA.

The decidability of the equational theory of KAT is obtained through its reduction to PDL. KAT equational theory is *PSPACE*-complete, the same complexity for deciding KA equational theory [19].

2.3. Hoare Logic

Hoare logic (HL) [6] is a formal system for reasoning about the correctness of programs. It is based on the notion that a specification of a program considers *pre-conditions* and *post-conditions*, in the sense that:

1. The pre-condition is a predicate that defines which properties of the state should be true in order for the program to execute correctly. If the implementation of the specification does not respect the pre-condition, then the program can behave in unexpected ways.

2. The post-condition is a predicate describing the conditions that the program establishes after a correct execution.

The implementation of the specification can be either *partial* or *total*, in the sense that the former does not guarantee that the program terminates, and the latter requires that the program does terminate.

HL uses *Hoare Triples* to reason about program correctness. An Hoare triple, also known by *partial correctness assertion* (PCA for short), is of the form

$$\{P\}S\{Q\} \tag{39}$$

where P is the pre-condition, Q is the post-condition and S is a program. Total correctness means we start in a state where P is true and we execute S, then S will terminate in a state satisfying Q. Partial correctness guarantees that Q is satisfied only if the program terminates, which may not be the case.

HL provides a set of rules for deriving valid PCA's. A first-order version of the HL rules for the partial correctness of a simple *while*-language with assignment are given bellow. The reader should direct to [6] for a detailed description of each of these rules. Here, we just briefly describe them.

$$\frac{}{\{P[x/E]\}\, x := E\, \{P\}}\ \text{(Assignment rule)}$$

$$\frac{\{P\}\, S\, \{Q\} \qquad \{Q\}\, T\, \{R\}}{\{P\}\, S; T\, \{R\}}\ \text{(Composition rule)}$$

$$\frac{\{B \wedge P\}\, S\, \{Q\} \qquad \{\neg B \wedge P\}\, T\, \{Q\}}{\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \{Q\}}\ \text{(Conditional rule)}$$

$$\frac{\{P \wedge B\}\, S\, \{P\}}{\{P\}\ \mathsf{while}\, B\, \mathsf{do}\, S\ \{\neg B \wedge P\}}\ \text{(While rule)}$$

$$\frac{P \to P' \qquad \{P'\}\, S\, \{Q'\} \qquad Q' \to Q}{\{P\}\, S\, \{Q\}}\ \text{(Weakening rule)}$$

The rules of *composition*, *conditional* and *weakening* are simple. The composition rule states that the post-condition of the program S must be the pre-condition of the program T when executing them sequentially. The conditional rule states that the post-condition of a conditional statement is the same for both the *if-then-else* programs S and T. The weakening rule states that we can always strengthen a pre-condition, and weaken a post-condition.

The rules for variable assignment and while loops are usually more difficult to interpret. The *assignment* rule states that if we want to show that P holds after the assignment of E to the variable x, then we must show that $P[x/E]$ (the substitution of the free occurrences of x by E, in P) holds before the assignment. The *while* rule is the more complicated one, since we have to deal with the invariant P which must be true in each iteration of the while loop. This rule dictates that when B is false, that is, when the loop condition fails, the invariant must be true, no matter how many time the loop has been repeated before.

2.4. Encoding PHL in KAT

By reducing HL to a propositional level, by omitting the assignment statements in programs, we get the *propositional Hoare logic* (PHL) [7,20,21], a formal system which allows the specification and verification of static assertions about the underlying domain of computation. We will see that KAT subsumes PHL, *i.e.*, that PHL deduction rules become theorems of KAT and therefore, deductions in PHL become pure equational reasoning in KAT. The deduction rules of PHL are obtained by not considering the assignment rule of HL. These deduction rules can be written as KAT equations, using the representation of Hoare triples, or PCA's, in KAT.

A PCA in PHL has the form $\{b\}p\{c\}$, where b and c are propositional variables and p is a program. The variable b states a pre-condition, and c states the corresponding post-condition, after p is executed. In KAT we write the PCA $\{b\}p\{c\}$ as follows:

$$bp\ \bar{c} = 0 \tag{40}$$

or, equivalently as the equation

$$bp = bpc \tag{41}$$

The equivalence of both terms is easily proved as follows:

$$bp = bp(c + \bar{c}) = bpc + bp\ \bar{c} = bpc + 0 = bpc$$

and:

$$bp\ \bar{c} = bpc\ \bar{c} = bp0 = 0$$

As we have already stated, the representation of PHL deduction rules become KAT expressions. Their usage in program verification reduces to pure equational reasoning. The PHL deduction rules are written in KAT as follows:

1. *Composition rule in KAT:*

$$bp = bpc \wedge cq = cqd \rightarrow bpq = bpqd \tag{42}$$

2. *Conditional rule in KAT:*

$$bcp = bcpd \wedge \bar{b}cq = \bar{b}cqd \rightarrow c(bp + \bar{b}q) = c(bp + \bar{b}q)d \tag{43}$$

3. *While rule in KAT:*

$$bcp = bcpc \rightarrow c(bp)^* \,\bar{b}\} = c(bp)^* \,\bar{b}\,\bar{b}c \tag{44}$$

4. *Weakening rule in KAT:*

$$b' \leq b \wedge bp = bpc \wedge c \leq c' \rightarrow b'p = b'pc' \tag{45}$$

To establish the adequacy of these translations we present the following theorem, properly proved by Dexter Kozen in [8]:

Theorem 1. *The formulas (42-45) are theorems of KAT.*

In this way we know that any rule of the form

$$\frac{\{b_1\}p_1\{c_1\} \qquad \ldots \qquad \{b_n\}p_n\{c_n\}}{\{b\}p\{c\}}$$

which can be derived in PHL has a KAT counterpart corresponding to the equational implication (universal Horn formula)

$$b_1p_1\bar{c}_1 = 0 \wedge \ldots \wedge b_np_n\bar{c}_n = 0 \rightarrow bp\bar{c} = 0$$

that is a theorem of KAT.

3. The Coq theorem prover

The Coq interactive theorem prover [10] is an implementation of the *Calculus of Inductive Constructions* (CIC for short) [22], a typed λ-calculus with a primitive notion of *inductive types*. An inductive type is a collection of *constructors*, each with its own arity. Each inductive definition also comes with an elimination principle. These elimination principles of Coq are splitted into case analysis, and fixpoint guarded by structural decreasing.

Coq is based on the *Curry-Howard isomorphism* principle, that is, any typing relation $t : A$ can either be seen as stating that t has type A, or as stating that t is a proof of the proposition A. Any type in Coq is of one of three kinds of *sorts*: *Set*, *Prop* and *Type*. The first two correspond to the informational and logical terms, respectively. Both belong to the *Type* sort.

Working with Coq consists of users introducing definition, variables, axioms, etc. in the system's proof-context, and make proofs in *natural* *deduction* style, where each proof step is mechanically checked by the system. In order to facilitate the construction of the proofs, Coq has available a set of *tactics* which deal with common proof tasks that usually a user must apply in building its proofs.

For a deeper understanding of the theory behind Coq, and about the way one can build proofs in it, the reader should direct to [10].

4. KAT and PHL in Coq

The encoding of both KAT and PHL in Coq closely follows the theory we have presented in the previous section. We have implemented these theories using Coq's module system and based on the encoding of commutative rings by Grégoire and Mahboubi in [23]. We have developed module signatures and modules for the Boolean algebra and for the Kleene algebra parts. Based on these implementations we implemented a module for KAT.

The encoding of PHL was done as presented in Section 2.4, as another Coq module importing the KAT module, and proving that the rules of PHL are theorems of KAT.

Our implementation intends to be a *proof of concept* that KAT can indeed be applied to program verification, and to the production of proof certificates that assure the partial correctness of simple *while*-programs. The application consists of Coq modules encoding the different algebraic structures, the deduction of PHL, and a set of examples that will be the subject of Section 5.

4.1. Encoding the sets of axioms

Both BA and KA are idempotent semirings equipped with extra operations to represent, respectively, Boolean negation and Kleene's closure. Therefore, they share the semiring axioms and each of them have extra sets of axioms that describe the properties of the extra operators. The operators and constants of a semiring are denoted as usual, by declaring in Coq notations and infixes as appropriated. Boolean negation is represented as ~x, and Kleene's star as x#.

We have coded the various sets of axioms in the same spirit of [23], by defining them as Coq *records*.

```
Variable U : Set.
Variable Uzero Uone : U.
Variable Uneg Ustar : U -> U.
Variable Uplus Udot : U -> U -> U.
Variable Ueq Uleq   : U -> U -> Prop.

(* Idempotent semiring axiom set *)
Record idemp_semi_ring_theory : Prop := mk_isr {
        isr_dot_zero_l  : forall x,     0 * x = 0 ;
        isr_dot_zero_comm : forall x,   0 * x = x * 0 ;
        isr_plus_zero_l : forall x,     0 + x = x ;
        isr_dot_one_l   : forall x,     1 * x = x ;
        isr_dot_one_comm : forall x,    1 * x = x * 1 ;
        isr_plus_idem   : forall x,     x + x = x ;
        isr_plus_comm   : forall x y,   x + y = y + x ;
        isr_plus_assoc  : forall x y z, (x + y) + z = x + (y + z) ;
        isr_dot_assoc   : forall x y z, (x * y) * z = x * (y * z) ;
        isr_dot_distr_l : forall x y z, x * (y + z) = x * y + x * z ;
        isr_dot_distr_r : forall x y z, (y + z) * x = y * x + z * x
}.

(* Boolean algebra axiom set *)
Record boolean_theory  : Prop := mk_bool {
        ba_ax_1 : forall x,   x * ~x = 0 ;
        ba_ax_2 : forall x,   x + ~x = 1 ;
        ba_ax_3 : forall x,   x + 1 = 1 ;
        ba_ax_4 : forall x y, ~(x + y) = ~x * ~y ;
        ba_ax_5 : forall x y, ~(x * y) = ~x + ~y ;
        ba_ax_6 : forall x,   ~~x = x ;
        ba_ax_7 : forall x y, x * y = y * x ;
        ba_ax_8 : forall x,   x * x  = x
}.

(* Kleene's close axiom set *)
Record ka_theory : Prop := mk_ka {
        star_ax_1 : forall x,     1 + (x * x#) = x# ;
        star_ax_2 : forall x,     1 + (x# * x) = x# ;
        star_ax_3 : forall x y z, ((z + y * x) <= x) -> (y# * z <= x) ;
        star_ax_4 : forall x y z, ((z + x * y) <= x) -> (z * y# <=  x)
}.
```

In the proofs we have mostly used Coq's *rewrite* tactic, which is natural for reasoning about equalities. In short words, this tactic rewrites sub-terms using equalities already available in the proof context of Coq. For instance, if we have a term `X` of the type `a=b` and we have a goal with type `P(a)`, then applying `rewrite X` to the goal yields a goal of type `P(b)`.

Another tactic often used in our proofs is the *pattern* tactic, which allowed us to apply rewriting to specific occurrences of sub-terms in the hole equality whenever the rewriting of all the occurrences of such sub-term is not desired.

4.2. Implementation of BA in Coq

The encoding of the BA in Coq was our first task. This encoding consisted in the implementation of a module signature for the BA, and the implementation of a module which takes another Coq module satisfying the signature of a BA and proving extra properties about BA's. Below we present part of the BA module, namely the proof that $\forall b\ c\ d \in B,\ b + cd = (b + c)(b + d)$ is a theorem of any BA.

```
Module Type BA_sig.

Parameter B : Set.

Parameter B0 B1 : B.
Parameter Bneg  : B -> B.
Parameter Bplus Bdot : B -> B -> B.
Parameter Is_idemp_semi_ring : idemp_semi_ring_theory B0 B1 Bplus Bdot.

Parameter Is_boolean : boolean_theory B0 B1 Bneg Bplus Bdot.

End BA_sig.
```

```
Module BA ( ba : BA_sig).

Import ba.
Export ba.

(* Renaming of semiring axioms *)
Definition plus_zero_l   := (isr_plus_zero_l Is_idemp_semi_ring).
Definition plus_comm     := (isr_plus_comm Is_idemp_semi_ring).
Definition plus_assoc    := (isr_plus_assoc Is_idemp_semi_ring).
Definition dot_one_l     := (isr_dot_one_l Is_idemp_semi_ring).
Definition dot_one_comm  := (isr_dot_one_comm Is_idemp_semi_ring).
Definition dot_zero_l    := (isr_dot_zero_l Is_idemp_semi_ring).
Definition dot_zero_comm := (isr_dot_zero_comm Is_idemp_semi_ring).
Definition plus_idem     := (isr_plus_idem Is_idemp_semi_ring).
Definition dot_assoc     := (isr_dot_assoc Is_idemp_semi_ring).
Definition dot_distr_l   := (isr_dot_distr_l Is_idemp_semi_ring).
Definition dot_distr_r   := (isr_dot_distr_r Is_idemp_semi_ring).


(* Derived and important results *)
Lemma plus_distr : forall b c d, b + (c * d) = (b + c) * (b + d).
Proof.
    intros.
    rewrite dot_distr_r.
    repeat rewrite dot_distr_l.
    rewrite ba_ax_8.
    rewrite <- (dot_one_r b);rewrite dot_assoc.
    rewrite <- dot_distr_l.
    rewrite (plus_comm 1 (1 * d)).
    rewrite ba_ax_3.
    rewrite dot_one_r.
    rewrite <- plus_assoc.
    pattern b at 2.
    rewrite <- dot_one_l.
    rewrite <- dot_distr_r.
    pattern (1+c) at 1;rewrite plus_comm.
    rewrite ba_ax_3.
    rewrite dot_one_l.
    reflexivity.
Qed.

End BA.
```

4.3. Implementation of KA in Coq

The encoding of KA in Coq was the second step of the overall work. We have followed the same approach as we did when coding the BA, that is, we have coded a signature of a KA and also a module with the proofs of a set of properties which are intensively used when reasoning about KA equalities.

The KA signature, named `KA_sig`, was encoded as follows.

```
Module Type KA_sig.

Parameter K : Set.

Parameter K0 K1      : K.
Parameter Kstar      : K -> K.
Parameter Kplus Kdot : K -> K -> K.
```

```
Definition Kleq (x y:K) := Kplus x y = y.

Parameter Is_idemp_semi_ring : idemp_semi_ring_theory K0 K1 Kplus Kdot.

Parameter Is_ka : ka_theory K1 Kstar Kplus Kdot Kleq.

End KA_sig.
```

The KA module receives as a parameter another module satisfying the signature of a KA and contains proofs of many important properties which are used when reasoning about KA equalities. Here we present some of these proofs, namely that $\forall$ x y, $x \le y \to x^* \le y^*$ (property (22), here denoted as `Le_Mon_Star`) and □ x y, $(xy)^*x = x(yx)^*$ (property (28), here named `Sliding`) from Section 2.1.

```
Module KA ( ka : KA_sig ).

Import ka.
Export ka.

(* Renaming of semiring axioms *)
Definition plus_zero_     := (isr_plus_zero_l Is_idemp_semi_ring).
Definition plus_comm      := (isr_plus_comm Is_idemp_semi_ring).
Definition plus_assoc     := (isr_plus_assoc Is_idemp_semi_ring).
Definition dot_one_l      := (isr_dot_one_l Is_idemp_semi_ring).
Definition dot_one_comm   := (isr_dot_one_comm Is_idemp_semi_ring).
Definition dot_zero_l     := (isr_dot_zero_l Is_idemp_semi_ring).
Definition dot_zero_comm  := (isr_dot_zero_comm Is_idemp_semi_ring).
Definition plus_idem      := (isr_plus_idem Is_idemp_semi_ring).
Definition dot_assoc      := (isr_dot_assoc Is_idemp_semi_ring).
Definition dot_distr_l    := (isr_dot_distr_l Is_idemp_semi_ring).
Definition dot_distr_r    := (isr_dot_distr_r Is_idemp_semi_ring).



Theorem Le_Mon_Star : forall x y, x <= y -> x# <= y#.
Proof.
intros.
rewrite <- (star_ax_1 y).
rewrite <- (dot_one_r (x#)).
apply star_ax_3.
pattern (1 + y*y#) at 1.
rewrite star_ax_1.
apply Le_Mon_Plus_Left.
apply Le_Mon_Dot_Right.
assumption.
Qed.

Theorem Sliding : forall x y, (x * y)# * x = x * (y * x)#.
Proof.
symmetry.
apply Bissimulation.
rewrite <- dot_assoc.
reflexivity.
Qed.

End KA.
```

4.4. Implementation of KAT in Coq

The implementation of KAT was achieved by the coding of a new Coq module, named KAT, which takes two arguments: a module satisfying the signature of a BA, and a module satisfying the signature of a KA. As defined in Section 2.2, the BA is embedded in the KA, which is stated by the condition $B \subseteq K$, where B is the set of BA terms and K is the set of KA terms. This set inclusion was implemented by the specification of a coercion named `ba_to_ka` which transforms terms of a BA in terms of a KA. Furthermore, we establish four parameters which refer to the properties that such a coercion should have, namely the fact that the constants 0 and 1, and the operations $+$ and $\cdot$ are the same in both BA and KA.

```
Module KAT (ka : KA_sig) (ba : BA_sig ).

Module Ka := KA ka.
Module Ba := BA ba.

Import Ka.
Export Ka.

(* Coercion from B to K *)
Parameter ba_to_ka : ba.B  -> K.
Coercion  ba_to_ka : ba.B >-> K.
(* Properties that the coercion must exhibit *)
Parameter ba_B1_eq_ka_K1 : ba_to_ka ba.B1 = 1.
Parameter ba_B0_eq_ka_K0 : ba_to_ka ba.B0 = 0.
Parameter ba_Bplus_eq_ka_Kplus :
        forall x y, ba_to_ka (ba.Bplus x y) = ba_to_ka x + ba_to_ka y.
Parameter ba_Bdot_eq_ka_Kdot:
        forall x y, ba_to_ka (ba.Bdot x y)  = ba_to_ka x * ba_to_ka y.
```

Besides establishing this coercion between elements of BA and KA, we prove that the axioms of the BA are also axioms of the KAT. To build the proofs we coded a simple tactic which can be applied to each of the BA axioms and prove it in the KAT module. Here we present the tactic and apply it to one of the axioms as an example.

```
Ltac solve_ba_ax x := intros;rewrite x;reflexivity.

Theorem ba_ax_1 : forall x:ba.B, ba_to_ka (x * ~x) = ba_to_ka (0).
Proof.
 solve_ba_ax Ba.ba_ax_1.
Qed.

End KAT.
```

5. Implementation of PHL in Coq

The implementation of PHL reflects the encoding of the PHL rules as KAT theorems, as presented in Section 2.4. Thus, we have coded a new module named PHL where we import the KAT module and use it to prove that PHL's deduction rules are theorems of KAT.

Next we encode the PCA and prove that the two possible representations for the PCA are equivalent.

```
Module PHL.

Declare Module ka : KA_sig.
Declare Module ba : BA_sig.

Module kat := KAT ka ba.

Import kat.

(* Partial Correctness Assertion *)
Definition PCA (b c : ba.B) (x : K) := b * x * ~c = 0.

Section Partial_Correctness_Assertion.

(* Equivalent representaion of PCA condition *)
Lemma PCA_Equiv_1 : forall b c : ba.B , forall x : K, PCA b c x -> b * x = b * x *
c.
Proof.
 intros.
 rewrite dot_assoc.
 rewrite <- (dot_one_r (b*x)).
 rewrite <- ba_B1_eq_ka_K1.
 rewrite <- (ba_ax_2 c).
 rewrite ba_Bplus_eq_ka_Kplus.
 rewrite dot_distr_l.
 rewrite H.
 rewrite plus_zero_r.
 rewrite dot_assoc.
 reflexivity.
Qed.

Lemma PCA_Equiv_2 : forall b c : ba.B, forall x, b * x = b * x * c -> PCA b c x.
Proof.
 intros.
 unfold PCA.
 rewrite H.
 repeat rewrite dot_assoc.
 rewrite <- ba_Bdot_eq_ka_Kdot.
 rewrite  ba_ax_1.
 rewrite ba_B0_eq_ka_K0.
 repeat rewrite dot_zero_r.
 reflexivity.
Qed.

End Partial Correctness Assertion.
```

The proofs for the *composition* rule and for the *conditional* rule are as follows:

```
Section Composition_Rule.

Theorem Composition : forall b c d : ba.B, forall x y,
        b * x = b * x * c /\ c * y = c * y * d
                         ->
                b * x * y = b * x * y * d.
Proof.
 intros b c d x y H.
 elim H.
 intros H0 H1.
 rewrite H0.
 rewrite dot_assoc.
 rewrite H1.
 repeat rewrite dot_assoc.
 rewrite <- (ba_Bdot_eq_ka_Kdot d d).
 rewrite ba_ax_8.
 reflexivity.
Qed.

End Composition_Rule.
```

```
Section  Conditional_Rule.

Theorem Conditional : forall b c d : ba.B, forall  x y,
        b * c * x = b * c * x * d /\ ~b * c * y  = ~b * c * y * d
                                   ->
          c * (b * x + ~b * y) = c * (b * x + ~b * y) * d.
Proof.
 intros b c d x y H.
 elim H.
 intros H0 H1.
 rewrite dot_distr_l.
 repeat rewrite  <- dot_assoc.
 rewrite <- ba_Bdot_eq_ka_Kdot.
 rewrite (ba_ax_7 c b).
 rewrite dot_distr_r.
 rewrite ba_Bdot_eq_ka_Kdot.
 rewrite H0.
 rewrite (dot_assoc (b*c*x) d d).
 rewrite <- (ba_Bdot_eq_ka_Kdot d d).
 rewrite ba_ax_8.
 repeat rewrite <- ba_Bdot_eq_ka_Kdot.
 rewrite (ba_ax_7 c ~b).
 repeat rewrite ba_Bdot_eq_ka_Kdot.
 rewrite H1.
 rewrite (dot_assoc (~b*c*y) d d).
 repeat rewrite <- (ba_Bdot_eq_ka_Kdot d d).
 rewrite ba_ax_8.
 reflexivity.
Qed.

End Conditional_Rule.
```

6. Verification of programs

In this section we provide examples of formal proofs of simple *while*-programs, using our implementation of KAT and PHL in the Coq theorem prover. All the *while*-language constructs will be considered, including assignments. However, since assertions are not allowed in KAT, we have to transform them into KAT terms, which serve as hypothesis for the equational reasoning. This approach was used by Dexter Kozen and others in the application of KAT and PHL in several verification tasks [5,11].

6.1. Program composition

Consider the following while-program:

```
y := x;
y := x + x + y;
```

What we intend to prove about this program is that the value of y is going to be three times the value x, independently of the value that x assumes initially. The pre-condition for the program is the *true* predicate, and the post-condition is $y = 3x$. The corresponding PCA is

$$\{true\}\ y := x\ ;\ y := x + x + y\ \{y = 3x\} \tag{46}$$

Its correctness proof can be obtained by first order Hoare logic reasoning, as presented in Figure 1. For building the proof, we used the tableau proof system for HL described in [24] and based on the computation of *weakest preconditions*.

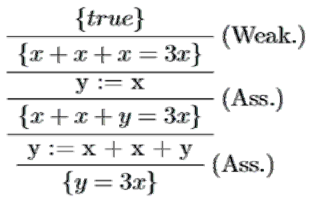

Fig. 1. Tableau for the correctness of the PCA (46)

The encoding of the PCA (46) is obtained by assigning propositional variables to the atomic commands, and to the atomic predicates of the condition. Assuming that:

- $p =_{def}$ "$y := x$"
- $q =_{def}$ "$y := x + x + y$"
- $b =_{def} 1$
- $c =_{def}$ "$y = 3x$"

the PCA is written in KAT as

$$bpq = bpqc \tag{47}$$

We must consider as hypothesis the instances of the *assignment* rule. Let $d =_{\text{def}}$ "$x + x + y = 3x$" then we have:

$$1p = 1pd \Leftrightarrow \text{``}\{x + x + x = 3x\}\ y := x\ \{x + x + y = 3x\}\text{"}$$

$$dq = dqc \Leftrightarrow \text{`}\{x + x + y = 3x\}\ y := x + x + y\ \{y = 3x\}\text{"}$$

Considering the hypothesis and the composition rule, we derive the PCA (47). The proof is written in Coq as follows:

```
Variable b c d : ba.B.
Variable p q   : K.

Theorem With_Composition : b = 1 -> p = p*c -> c*q = c*q*d ->  b*p*q = b*p*q*d.
  Proof.
    intros b c d p q H H0 H1.
    eapply Composition.
    split.
    rewrite H.
    repeat rewrite <- dot_one_comm.
    repeat rewrite dot_one_r.
    apply H0.
    exact H1.
  Qed.
```

6.2. Conditional

Consider the following *while*-program for calculating the maximum between two numbers, which we denote by *Cond*:

```
if x >= y then
  MAX := x
else
  MAX := y
```

We want to show that *{true}Cond{MAX=max(x,y)}* is valid. We know that $x \geq y \rightarrow x = max(x,y)$ and that $\neg(x \geq y) \rightarrow y = max(x,y)$. Using these facts, we can build the correctness tableau of the PCA as presented in Figure 2. In the tableau we consider the following derived rule for the conditional statement:

$$\frac{\{P_1\}\,S\,\{Q\} \qquad \{P_2\}\,T\,\{Q\}}{\{(B \rightarrow P_1) \wedge (\neg B \rightarrow P_2)\}\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \ \{Q\}}\ (\textbf{Conditional rule II})$$

$$\begin{array}{c l}
\{true\} & \\ \hline
\textbf{if}\ \mathrm{x} \geq \mathrm{y}\ \textbf{then} & (\text{Cond.}) \\ \hline
\{max(x,y) = x\} & \\ \hline
\mathrm{MAX} := \mathrm{x} & (\text{Ass}) \\ \hline
\{MAX = max(x,y)\} & \\ \hline
\textbf{else} & (\text{Cond.}) \\ \hline
\{y = max(x,y)\} & \\ \hline
\mathrm{MAX} := \mathrm{y} & (\text{Ass.}) \\ \hline
\{MAX = max(x,y)\} & (\text{Cond.}) \\ \hline
\{MAX = max(x,y)\} &
\end{array}$$

Fig. 2. Correctness proof of the *Cond* program PCA.

We now proceed by asserting meanings to KAT variables:

- $b =_{def}$ "$x \geq y$"
- $\overline{b} =_{def}$ "$\neg(x \geq y)$"
- $c =_{def}$ "$MAX = max(x,y)$"
- $p =_{def}$ "$MAX := x$"
- $q =_{def}$ "$MAX := y$"

Defining the PCA of the *Cond* program in PHL, we obtain:

$$1(bp + \overline{b}q)c$$

By assuming the assertions $d =_{def}$ "$max(x,y) = x$" and $e =_{def}$ "$max(x,y) = y$", and using as hypothesis

$$dp = dpc \Leftrightarrow \{max(x,y) = x\}\ MAX := x\ \{MAX = max(x,y)\}$$

$$ep = epc \Leftrightarrow \{max(x,y) = y\}\ MAX := y\ \{MAX = max(x,y)\},$$

the Coq correctness proof follows directly from the application of the conditional rule, from rewriting of the hypothesis, and from Boolean reasoning. Here we omit its script.

6.3. While

As a final example, we verify a *while*-program containing a while loop, to which we call *Whl*. Its code is:

```
a := 0;
z := 0;
while (a != y)
  z := z + x;
  a := a + 1
```

Having as pre-condition the *true* predicate, and as post-condition the predicate $z = xy$, we build its correctness tableau as presented in Figure 3. In this case the invariant is $z=xa$.

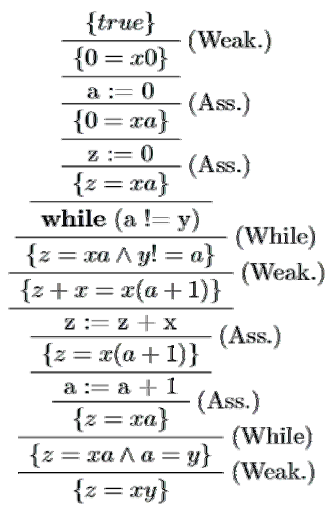

Fig. 3. Correctness proof of the *Whl* program

We assert the following meanings to KAT variables:

- $b =_{def}$ "a != y"
- $c =_{def}$ "$z = xy$"
- $p =_{def}$ "$a := 0$"
- $q =_{def}$ "$z := 0$"
- $r =_{def}$ "$z := z + x$"
- $s =_{def}$ "$a := a + 1$"

The definition of the PCA of *Whl* is:

$$1(pq)((b(rs))^*\ \bar{b}) = 1(pq)((b(rs))^*\ \bar{b})c$$

Using the information provided by the tableau, and considering the assertions

- $e =_{def}$ "$z = xa$"
- $g =_{def}$ "$0 = xa$"
- $b_0 =_{def}$ "$z + x = x(a + 1)$"
- $b_1 =_{def}$ "$z = x(a + 1)$",

we draw the following hypothesis:

$\bar{b}e = c$	$\Leftrightarrow$	``$z = xa \wedge a = y \rightarrow z = xy$"
$bc \leq b_0$	$\Leftrightarrow$	"$z = xa \wedge \neg(a = y) \rightarrow z + x = x(a + 1)$
$dp = dpg$	$\Leftrightarrow$	``$\{0 = x0\}\ a := 0\ \{0 = xa\}$"
$gq = gqe$	$\Leftrightarrow$	``$\{0 = xa\}\ z := 0\ \{z = xa\}$"

$$b_0r = b_0rb_1 \quad \Leftrightarrow \quad ``\{z + x = x(a + 1)\}\ z := z + x\ \{ z = x(a + 1)\}"$$
$$b_1s = b_1se \quad \Leftrightarrow \quad ``\{ z = x(a + 1) \}\ a := a + 1\ \{z = xa\}"$$

In Coq, by using the previous equalities, by applying twice the composition rule, and by applying the while rule, we are left with simple KAT manipulation to finish our proof. The proof script for the *Whl* program is the following:

```
Variables p q r s : K.
Variables b d c b0 b1 b2 b3 b4: ba.B.

Hypothesis hip_1 : d = B1.
Hypothesis hip_2 : d * p = d * p * g.
Hypothesis hip_3 : g * q = g * q * e.
Hypothesis hip_4 : ~b*e = c.
Hypothesis hip_5 : c * b <= b3.
Hypothesis hip_6 : b0 * r = b0 * r * b1.
Hypothesis hip_7 : b1 * s = b1 * s * e.

Theorem With_While : d*(p*q)*((b*(r*s))#*~b) = d*(p*q)*((b*(r*s))#*~b)*c.
Proof.
 apply Composition with (c:=e).
 intuition.
 repeat rewrite <- dot_assoc.
 eapply Composition.
 split.
 rewrite <- hip_1.
 apply hip_2.
 assumption.
 rewrite <- hip_4.
 repeat rewrite <- dot_assoc.
 rewrite (dot_assoc b r s).
 apply While.
 rewrite <- hip_8.
 inversion hip_5.
 inversion hip_5.
Qed.
```

7. Comments and applications

Currently, our implementation contains no automation for building proofs in KAT, except for the default tactics provided by Coq. Also, the verification of programs can only be done after we encode the programs in KAT by hand.

First-order assignments $x{:=}t$ (and Hoare logic's assignment rule) can be considered if we extend KAT to Schematic KAT (SKAT), a version of KAT whose intented semantics coincides with the semantics of flowchart schemes over a ranked alphabet [5,25,26]. We have followed these ideas when building the KAT for the programs verified but, however, that was not done automatically. The usage of SKAT allows for the automatic production of assertions relating assignments to KAT variables, which will lead us to a more automated implementation.

We intend to improve the current implementation in order to obtain more compact proof scripts, and also to identify proof patterns along the scripts which can be replaced by Coq tactics implemented by ourselves. We also aim at building tactics for the automatic production of proofs for KAT equalities, possibly based on the recent works of Hubie Chen and Riccardo Pucella [27], James Worthington [28], and Dexter Kozen [29], that presented approaches to determine automatically the equivalence between KAT expressions.

We have developed a set of Coq scripts that formally implement the KAT theory and shown its application to program verification using PHL. We did this with the aim of studying the feasibility of using KAT in the context of enforcement of security mechanisms for Embedded Systems, in particular, *Proof-Carrying-Code* systems.

Proof-Carrying-Code aims at providing static and decentralized security enforcement mechanisms based on the notion of verifiable evidence, usually defined as certificate. The key idea of Proof-Carrying-Code is to attach to a (mobile) code file, an easily to check proof (certificate) that its execution does not violate certain safety policies.

One possibility is to use KAT as the formal system to write certificates for programs, even at the source code level of the application. KAT has a very compact representation and there exists automatic procedures to decide KAT equations. This means that we can automate the production of certificates in Coq by implementing a tactic which automatically proves KAT properties which we might be interested in the context of mobile code security. The computational cost of the implementation of such a system will be subject to future research.

8. Conclusions

In this document we have presented an implementation of KAT in the Coq theorem prover. We also shown how the implementation of the KAT in Coq served as a framework for verifying simple *while*-programs, using the PHL rules and KAT reasoning.

As we have seen, simple imperative languages have somehow direct translations into KAT expressions. Moreover, KAT is very compact and needs small computation power for being analysed and processed for proof correctness. This aspect is fundamental when considering Embedded Systems, which are now more in the mobile systems concept and with considerable limitations either in computational power and lack of energy for keeping alive and executing. The implementation we have presented here can be seen as the starting point for such a system.

For future work, we intend to augment the implementation we have, by proving more properties and develop a certified algorithm which decides if two KAT expressions are equivalent, in an automatic way. Such algorithm will facilitate the introduction of KAT in the realm of new computer paradigms, as is the case of Proof-Carrying-Code.

9. Acknowledgments

We are most grateful to Simão Melo de Sousa for motivating us for this work, and for providing us with comments and suggestions that were fundamental for this work. We also thank him for his help on using Coq properly in our implementation.

We thank very much to the referee that pointed us a problem of inconsistency in a previous version of our Coq development. We also thank to Jean-Christophe Filliâtre, and to the members of the RESCUE project.

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and program POSI, and the FCT RESCUE project PTDC/EIA/65862/2006. David Pereira is funded by FCT grant SFRH/BD/33233/2007.

10. References

1. Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems 19(3) (May 1997) 427–443
2. Kleene, S. In: Representation of Events in Nerve Nets and Finite Automata. Shannon, c. and Mccarthy, j. edn. Princeton University Press, Princeton, N.J. 3– 42
3. Barth, A., Kozen, D.: Equational verification of cache blocking in LU decomposition using Kleene algebra with tests. Technical Report TR2002-1865, Computer Science Department, Cornell University (June 2002)
4 Kozen, D., Patron, M.C.: Certification of compiler optimizations using Kleene algebra with tests. In: Computational Logic. (2000) 568–582
5. Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Cornell University (2001)
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10) (1969) 576–580
7. Kozen, D., Tiuryn, J.: On the completeness of propositional Hoare logic. In: RelMiCS. (2000) 195–202
8. Kozen, D.: On Hoare logic and Kleene algebra with tests. Trans. Computational Logic 1(1) (July 2000) 60–76
9. Necula, G.C.: Proof-carrying code. In: POPL ’97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1997) 106–119
10. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
11. Aboul-Hosn, K., Kozen, D.: KAT-ML: An interactive theorem prover for Kleene algebra with tests. Journal of Applied Non-Classical Logics 16(1–2) (2006) 9–33
12. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Infor. and Comput. 110(2) (May 1994) 366–390

13. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Theoretical Computer Science. North-Holland (1984)
14. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (November 2002)
15. Kozen, D.: Kleene algebra with tests and the static analysis of programs. Technical report, Cornell University (2003)
16. Aboul-Hosn, K., Kozen, D.: Local variable scoping and Kleene algebra with tests. In: RelMiCS. (2006) 78–90
17. Kozen, D., Smith, F.: Kleene algebra with tests: Completeness and decidability. In: CSL. (1996) 244–259
18. Kozen, D.: Automata on guarded strings and applications. Technical report, Cornell University, Ithaca, NY, USA (2001)
19. Kozen, D.: On the complexity of reasoning in Kleene algebra. Information and Computation 179 (2002) 152–162
20. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Transactions on Computational Logic (TOCL) 1(1) (2000) 60–76
21. Kozen, D.: On Hoare Logic, Kleene Algebra, and Types. Technical report, Cornell University (2000)
22. Paulin-Mohring, C.: Inductive definitions in the system coq: Rules and properties. Proceedings of the International Conference on Typed Lambda Calculi and Applications 664 (1993) 328–345
23. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in coq. In: Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS 3603, Springer (2005) 98–113
24. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press (2004)
25. Aboul-Hosn, K.: A Proof-Theoretic Approach to Mathematical Knowledge Management. PhD thesis, Cornell University (2007)
26. Aboul-Hosn, K.: An axiomatization of arrays for Kleene algebra with tests. In: Proc. 9th Int. Conf. Relational Methods in Computer Scienceand 4th Int. Workshop Applications of Kleene Algebra (RelMiCS/AKA'06). Volume 4136 of LNCS., Springer (2006) 63–77
27. Chen, H., Pucella, R.: A coalgebraic approach to Kleene algebra with tests. Theoretical Computer Science 327(1-2) (2004) 23–44
28. Worthington, J.: Automatic Proof Generation in Kleene Algebra. In: Relations and Kleene Algebra in Computer Science. Volume 4988 of LNCS. Springer Berlin / Heidelberg (2008) 382–396
29. Kozen, D.: On the coalgebraic theory of Kleene algebra with tests. Computing and information science technical reports, Cornell University (March 2008)

David Pereira received his B.S. and M.S. in Computer Science from the Faculty of Sciences of University of Porto, Portugal. Since 2003 he has worked as researcher in formal methods, first applied to Artificial Intelligence, and now in the context of the security of mobile code under the perspective of the Proof-Carrying-Code paradigm. He is currently an assistant researcher at LIACC-UP and a Computer Science Ph.D. student at the MAP-i doctoral program.

Nelma Moreira received her Ph.D. in Computer Science from the Faculty of Sciences of University of Porto, Portugal in 1997. She is a researcher at LIACC-UP and assistant professor at the department of Computer Science of the Faculty of Sciences of the University of Porto. Her research interests include the areas of automata theory and formal languages, and theorem proving and formal verification of software.

Received: July 16, 2008; Accepted: December 6, 2008.