Mechanization of an Algorithm for Deciding KAT Terms Equivalence[1]

Nelma Moreira David Pereira Simao Melo de Sousa

Technical Report Series: DCC-2012-04
Version 1.0 April 2012

U. PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Mechanization of an Algorithm for Deciding KAT Terms Equivalence

Nelma Moreira David Pereira † Simao Melo de Sousa
{nam,dpereira}@ncc.up.pt,desousa@di.ubi.pt

Abstract

This work presents a mechanically verified implementation of an algorithm for deciding the (in-)equivalence of Kleene algebra with tests (KAT) terms. This mechanization was carried out in the Coq proof assistant. The algorithm decides KAT terms equivalence through an iterated process of testing the equivalence of their partial derivatives. It is a purely syntactical decision procedure and so, it does not construct the underlying automata. The motivation for this work comes from the possibility of using KAT encoding of propositional Hoare logic for reasoning about the partial correctness of imperative programs.

1 Introduction

Kleene algebra with tests (KAT) [Koz97, KS96] is an algebraic system that extends *Kleene algebra* (KA) [Kle], the algebra of regular expressions, with Boolean tests. KAT is specially fitted to capture and verify properties of simple imperative programas and, in particular, subsumes *propositional Hoare logic* (PHL) [Koz00, KT01] in the sense that PHL's deductive rules are KAT theorems, and that proving a program partially correct is tantamount to checking if two KAT terms are equivalent.

Although KAT can be applied in several verification tasks, there are few support tools for that purpose. Aboul-Hosn and Kozen developed KAT-ML [AHK06], an interactive theorem prover for reasoning about KAT that also provides support for reasoning about simple imperative programs through SKAT [AK01], an extension of KAT with assignments. HÃşfner and Struth [HS07] used the automated theorem prover Prover9/Mace4 [McC] to axiomatically encode (variants of) Kleene algebras and to do proof experiments about Hoare logic, dynamic logic, temporal logics, concurrency control, and termination analysis.

In this paper we present a mechanically verified implementation in the Coq proof assistant [The] of a procedure to decide KAT terms equivalence using derivatives. Derivatives for KAT were introduced by Kozen [Koz08], who also presented a coinductive decision procedure for KAT terms equivalence. To the best of our knowledge, the work we present is the first mechanically verified procedure for KAT term (in)equivalence, and is a inductive approach rather than a coinductive one. Moreover, since we have implemented the decision procedure

*This work was partially funded by the European Regional Development Fund through the programme COMPETE and by the FundaÃğÃčo para a CiÃłncia e Tecnologia (FCT) under project CANTE-PTDC/EIA-CCO/101904/2008.

†This work was partially funded by the European Regional Development Fund through the programme COMPETE and by the FundaÃğÃčo para a CiÃłncia e Tecnologia (FCT) under project CANTE-PTDC/EIA-CCO/101904/2008.

in the Coq proof assistant, we can extract the procedure as a functional program that is correct by construction and that can be used in third party software. This work is the continuation of a previous work that consisted on the mechanically verified implementation of a decision procedure based on the same criteria, but applied to regular expressions (KA) [MPdS11]. It is also a maturation of an abstract formalization of KAT in Coq [MP08] where proofs of some simple properties of imperative programs could be interactively performed.

Recently, several formalizations of KA within proof assistants appear in the literature [BP10, KN11, CS, Kom, MPdS11]. Although we can reduce KAT terms equivalence to KA terms equivalence [KS96], such an approach does not seem to be feasible for practical proposes. Thus here we propose a specialized procedure for KAT. However, since the method we have used for KA does not involve the construction of any kind of automata and relies only on comparison of expressions, its adaptation to KAT was greatly simplified. This is clearly an advantage and suggests the possibility of other extensions. In this case, the adaptation was non trivial and it also required an implementation of the underlying language theoretical model of KAT, a new proof of the finiteness of the set of partial derivatives of KAT terms and the development of a procedure to handle tests.

2 Kleene Algebra with Tests

A *Kleene algebra* (KA) is an algebraic structure $(K, +, \cdot, ^\star, 0, 1)$ with $(K, +, \cdot, 0, 1)$ is an idempotent semiring and where the operator * is characterized by the following set of axioms

$$\begin{array}{ll} 1 + pp^\star \leq p^\star & q + pr \leq r \to p^\star q \leq r \\ 1 + p^\star p \leq p^\star & q + rp \leq r \to pq^\star \leq r, \end{array} \tag{1}$$

where $x \leq y$ is defined by $x + y = y$. The standard models for KA include regular expressions over a finite alphabet, binary relations and square matrices over another KA.

A *Kleene algebra with tests* (KAT) is an extension of a KA that contains an embedded *Boolean algebra*. Therefore, a KAT is characterized by the same set of axioms of KA plus the axioms of Boolean algebra. Formally KAT is a algebraic structure $(K, T, +, \cdot, ^\star, ^-, 0, 1)$ such that :

- $(K, +, \cdot, ^\star, 0, 1)$ is a KA;
- $(T, +, \cdot, ^-, 0, 1)$ is a Boolean algebra ;
- $(T, +, \cdot, ^-, 0, 1)$ is a subalgebra of $(K, +, \cdot, ^\star, 0, 1)$.

Let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a finite set of *primitive tests* and let $\Sigma = \{p_1, \ldots, p_m\}$ be a finite set of *primitive actions*. A *test* t is inductively defined by the following grammar:

$$t \in \mathsf{TExp} \ ::= \ 0 \,|\, 1 \,|\, b \in \mathcal{B} \,|\, \bar{t} \,|\, t + t \,|\, t \cdot t_2.$$

A KAT *term* e is a regular expression extended with tests, and it is inductively defined by the following grammar:

$$e \in \mathsf{Exp} \ ::= \ p \in \Sigma \,|\, t \,|\, e + e \,|\, e \cdot e \,|\, e^\star.$$

As usual we omit the concatenation operator $\cdot$ in both tests and KAT terms.

An important application of KAT is the verification of simple imperative programs. KAT are expressive enough to encode the notions of sequence, conditional and iterative repetition of instructions. These notions are captured by the following definitions:

$$
\begin{array}{rcl}
e_1;e_2 & \stackrel{def}{=} & e_1e_2 \\
\textsf{if}\ t\ \textsf{then}\ e_1\ \textsf{else}\ e_2\ \textsf{fi} & \stackrel{def}{=} & (te_1)+(\bar{t}e_2) \\
\textsf{while}\ t\ \textsf{do}\ e\ \textsf{end} & \stackrel{def}{=} & (te)^{\star}\bar{t}
\end{array}
$$

In particular KAT subsumes *propositional Hoare logic* (PHL), a fragment of standard Hoare logic [Hoa69] that does not contain assignments. PHL Hoare triples of the standard form $\{t_1\}e\{t_2\}$ are encoded in KAT by the equality $t_1e = t_1et_2$ or, equivalently, by the equality $t_1e\overline{t_2} = 0$, with $e \in \textsf{Exp}$. Moreover, PHL deductive rules are theorems of KAT [KT01] and deductive reasoning in PHL is replaced by equational reasoning in KAT.

In the Coq development, tests and KAT terms are encoded by the inductive types test and kat presented below. The sets $\mathcal{B}$ and Σ are specified by the abstract parameters sigmaB and sigmaP, respectively, and the types of primitive programs and primitive tests correspond to the types bv and sy, respectively.

```
Parameter sy bv : Type.
Parameter sigmaP : set sy.
Parameter sigmaB : set bv.

Inductive test : Type :=
| ba0    : test
| ba1    : test
| baV    : bv → test
| baN    : test → test
| baAnd : test → test → test
| baOr   : test → test → test.

Inductive kat : Type :=
| kats   : sy→ kat
| katb   : test → kat
| katu   : kat → kat → kat
| katc   : kat → kat → kat
| katst : kat → kat.
```

3 Language Theoretical Model of KAT

Similarly to KA, the usual models for KAT include languages and relations. Here we consider the language theoretical model of sets of *guarded strings*, as introduced by Kozen in [Koz01].

3.1 Literals and Atoms

Let $\mathcal{B}$ be the set of primitive tests and let $\overline{\mathcal{B}} = \{\bar{b} \,|\, b \in \mathcal{B}\}$. The elements $l \in \mathcal{B} \cup \overline{\mathcal{B}}$ are called *literals*. An *atom* is a finite sequence of literals

$$\alpha \in \{l_1l_2 \dots l_n \,|\, l_i \in \mathcal{B} \cup \overline{\mathcal{B}}\},$$

where $n = |\mathcal{B}|$, *i.e.*, an atom can be seen as a truth assignment to the elements of $\mathcal{B}$. The set of all atoms, which we denote by At, corresponds to the set of all possible truth assignments for the elements of $\mathcal{B}$. Thus, there exists exactly $2^{|\mathcal{B}|}$ atoms. Let t be a test and let α be an atom. We write $\alpha \leq t$ if $\alpha \rightarrow t$ is a propositional tautology. Thus we always have either $\alpha \leq b$ or $\alpha \leq \overline{b}$.

In Coq we have defined an abstract specification of literals, of atoms, and of a function to compute $\alpha \leq b$. We keep these definitions abstract in order to allow users to choose the best way to represent and compute with atoms. Moreover, the actual structure of atoms does not interfere with the implementation and correctness of the decision procedure.

3.2 Guarded Strings and Languages

The standard language theoretical model of KAT are sets of regular languages of *guarded strings* [Koz01]. A *guarded string* is a sequence $x = \alpha_0 p_0 \alpha_1 p_1 \dots p_{(l-1)} \alpha_l$, represented by the type gs below, and where $l \geq 0$, $\alpha_i \in \mathsf{At}$, and $p_i \in \Sigma$. If x is a guarded string, we define $\mathsf{first}(x) = \alpha_0$ and $\mathsf{last}(x) = \alpha_l$. Given two guarded strings x and y we say that x and y are *compatible* if $\mathsf{last}(x) = \mathsf{first}(y)$. If two guarded strings x and y are compatible, the *fusion product* xy is the standard word concatenation but omitting the common atom. If the guarded strings x and y are not compatible the fusion product is undefined.

The Coq function fusion_prod implements the fusion product of two guarded strings x and y. Its arguments are the guarded strings x and y and a proof of their compatibility. Due to dependent pattern matching, in the recursive branch where $x = \alpha p x'$, the proof that x and y are compatible must be transformed into a proof that x' and y are also compatible so that fusion_prod type-checks. This is the role of the lemma compatible_tl.

```
Inductive gs : Type :=
|gs_end   : atom → gs
|gs_conc  : atom → sy → gs → gs.

Definition last(x:gs) : atom.
Definition first(x:gs) : atom.

Definition compatible (x y:gs) := last x = first y.

Lemma compatible_tl:
  ∀ (x y x':gs)(α:atom)(p:sy),
   ∀ (h:compatible x y)(l:x = gs_conc x p x'), compatible x' y.

Fixpoint fusion_prod x y (h:compatible x y) : gs :=
 match x as x' return x = x' → gs with
 |gs_end _ ⇒ fun (_:(x = gs_end _)) ⇒ y
 |gs_conc k s t ⇒ fun (h0:(x = gs_conc k s t)) ⇒
                          let h' := compatible_tl x y h k s t h0 in
                           gs_conc k s (fusion_prod t y h')
end (refl_equal x).
```

A *language* is a set of guarded strings over the alphabets $\mathcal{B}$ and Σ. We denoted languages by G, G_i, with $i \in \mathbb{N}$. The set of all guarded strings is denoted by GS. Given two languages G_1 and G_2 we define the set G_1G_2 as the set of all the fusion products xy such that $x \in G_1$ and $y \in G_2$. The *power* of a language G, denoted by G^n, is inductively defined by

$$\begin{array}{rcl} G^0 & = & \mathsf{At}, \\ G^{n+1} & = & GG^n. \end{array} \tag{2}$$

The *Kleene star* of a language G is, consequently, defined by

$$G^\star = \cup_{n\geq 0} G^n. \tag{3}$$

Languages are defined as Prop-type functions in Coq, that is, predicates over terms of type gs. Below we provide the definition of the type of languages gl and the definitions of concatenation, power and Kleene star of terms of type gl.

```
Definition gl := gs → Prop.

Inductive gl_conc(gl1 gl2:gl) : gl :=
|mkg_gl_conc : ∀ (x y:gl)(T:compatible x y),
    x ∈ gl1 → y ∈ gl2 → (fusion_prod x y T) ∈ (gl_conc gl1 gl2).

Fixpoint conc_gln(l:gl)(n:nat) : gl :=
 match n with
 | 0    ⇒ gl_eps | S m ⇒ gl_conc l (conc_gln l m)
 end.

Inductive gl_star(l:gl) : gl :=
|mk_gl_star : ∀ (n:nat)(g:gs), g ∈ (conc_gln l n) → g ∈ (gl_star l).

Definition gl_eq (gl1 gl2:gl) := Same_set _ gl1 gl2.
Notation "x == y" := (gl_eq x y).
```

The interpretation of KAT terms as languages is given by the function G that is inductively defined by

$$\begin{array}{lcl} \mathsf{G}(p) & = & \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}\}\ ,\ p \in \Sigma \\ \mathsf{G}(t) & = & \{\alpha \in \mathsf{At} \mid \alpha \leq t\}\ ,\ t \in T \\ \mathsf{G}(e_1 + e_2) & = & \mathsf{G}(e_1) \cup \mathsf{G}(e_2) \\ \mathsf{G}(e_1 e_2) & = & \mathsf{G}(e_1)\mathsf{G}(e_2) \\ \mathsf{G}(e^\star) & = & \cup_{n \geq 0} \mathsf{G}(e)^n. \end{array} \tag{4}$$

It is straightforward to conclude that $\mathsf{G}(1) = \mathsf{At}$ and that $\mathsf{G}(0) = \emptyset$. Moreover, a guarded strings x is itself a KAT term and its language is $\mathsf{G}(x) = \{x\}$. We extend the function G to a set S of KAT terms in the usual way by $\mathsf{G}(S) = \cup_{e \in S} \mathsf{G}(e)$. If e_1 and e_2 are two KAT terms, we say that e_1 and e_2 are equivalent, and write $e_1 \sim e_2$, if and only if $\mathsf{G}(e_1) = \mathsf{G}(e_2)$. The same applies to sets of KAT terms. If S_1 and S_2 are sets of KAT terms then $S_1 \sim S_2$ if and only if $\mathsf{G}(S_1) = \mathsf{G}(S_2)$. Moreover, if e is a KAT term and S is a set of KAT terms then $e \sim S$ if and only if $\mathsf{G}(e) = \mathsf{G}(S)$.

The *left-quotient* of a language $G \subseteq \mathsf{GS}$ wrt. to elements $\alpha p \in (\mathsf{At} \cdot \Sigma)$ is defined by

$$(\alpha p)^{-1}(G) = \{\, x \mid \alpha p x \in G\}. \tag{5}$$

The notion of left-quotient is trivially extended to sequences $w \in (\mathsf{At} \cdot \Sigma)^\star$ as follows

$$w^{-1}(G) = \{\, x \mid wx \in G\}. \tag{6}$$

In Coq we have the function kat2gl that implements the function G, and the inductive predicates LQ and LQw that implement, respectively, the left-quotients of a language.

```
Fixpoint kat2gl(e:kat) : gl :=
 match e with
 | kats x ⇒ gl_sy x
 | katb b ⇒ gl_atom b
 | katu e1 e2 ⇒ gl_union (kat2gl e1) (kat2gl e2)
 | katc e1 e2 ⇒ gl_conc (kat2gl e1) (kat2gl e2)
 | katst e' ⇒ gl_star (kat2gl e')
 end.
```

```
Inductive LQ (l:gl) : atom → sy → gl :=
|in_quo : ∀ (a:atom)(p:sy)(y:gs), (gs_conc a p y) ∈ l → y ∈ LQ l a p.

Inductive LQw (l:gl) : gstring → gl :=
|in_quow : ∀ (x w:gs)(T:compatible w x), (fusion_prod w x T) ∈ l → x ∈ LQw l w.
```

3.3 Partial Derivatives of KAT Terms

The notion of *derivative* of a KAT term was introduced by Dexter Kozen and is an extension of Brzozowski's derivatives [Brz64].

Definition 1 *Let $\alpha \in \mathsf{At}$ and let $t \in T$. The function $\varepsilon : \mathsf{At} \to \mathsf{Exp} \to \{0,1\}$ is inductively defined by*

$$\varepsilon_\alpha(p) = 0 \qquad \varepsilon_\alpha(t) = \begin{cases} 1, & \text{if } \alpha \leq t \\ 0, & \text{if } \alpha \not\leq t \end{cases}$$

$$\varepsilon_\alpha(e_1 + e_2) = \varepsilon_\alpha(e_1) + \varepsilon_\alpha(e_2) \qquad \varepsilon_\alpha(e_1 e_2) = \varepsilon_\alpha(e_1) \cdot \varepsilon_\alpha(e_2)$$

$$\varepsilon_\alpha(e^\star) = 1$$

where $+$ and $\cdot$ are interpreted as the Boolean operations of disjunction and conjunction, respectively. The function ε is extended to the set of all atoms At by

$$\mathsf{E}(e) = \{\alpha \in \mathsf{At} \mid \varepsilon_\alpha(e) = 1\}. \tag{7}$$

The next theorem shows the utility of the function ε.

Theorem 1 *Let $\alpha \in \mathsf{At}$ and let e be a KAT term. If $\varepsilon_\alpha(e) = 1$ then $\alpha \in \mathsf{G}(e)$. Otherwise, $\alpha \notin \mathsf{G}(e)$.*

Let S be a set of KAT terms and let e be a KAT term. We define the concatenation of S with e by $Se = \{e'e \mid e' \in S\}$ if $e \neq 0$ and $e \neq 1$, and $S0 = \emptyset$ and $S1 = S$, otherwise. Similarly, we define eS. The former operation corresponds to the function dsr in the Coq formalization.

Definition 2 (Partial derivative) *Let $\alpha p \in (\mathsf{At} \cdot \Sigma)$ and let e be a KAT term. The* set $\partial_{\alpha p}(e)$ of partial derivatives *of e wrt. to αp is inductively defined by*

$$\partial_{\alpha p}(t) = \emptyset \qquad \partial_{\alpha p}(q) = \begin{cases} \{1\}, & \text{if } p \equiv q \\ \emptyset, & \text{if } p \not\equiv q \end{cases}$$

$$\partial_{\alpha p}(e_1 + e_2) = \partial_{\alpha p}(e_1) \cup \partial_{\alpha p}(e_2) \qquad \partial_{\alpha p}(e^\star) = \partial_{\alpha p}(e)e^\star$$

$$\partial_{\alpha p}(e_1 e_2) = \begin{cases} \partial_{\alpha p}(e_1)e_2 \cup \partial_{\alpha p}(e_2), & \textit{if } \varepsilon_\alpha(e_1) = 1 \\ \partial_{\alpha p}(e_1)e_2, & \textit{if } \varepsilon_\alpha(e_2) = 0 \end{cases}$$

Partial derivatives of KAT terms can be naturally extended to sequences $w \in (\mathsf{At} \cdot \Sigma)^\star$ by $\partial_\epsilon(e) = \{e\}$, and by $\partial_{w(\alpha p)}(e) = \partial_{\alpha p}(\partial_w(e))$, where ϵ is the empty sequence. The set of all partial derivatives of a KAT term e is the set

$$\partial_{(\mathsf{At} \cdot \Sigma)^\star}(e) = \bigcup_{w \in (\mathsf{At} \cdot \Sigma)^\star} \{e' \mid e' \in \partial_w(e)\}. \tag{8}$$

Partial derivatives are related to left-quotients as follows.

Theorem 2 *Let e be a* KAT *term, and let be a word $w \in (\mathsf{At} \cdot \Sigma)$. It holds that*

$$\mathsf{G}(\partial_w(e)) = w^{-1}(\mathsf{G}(e)).$$

The following excerpt of the Coq development shows the previous definitions and theorem. The function SkatL gives the language of a finite set of KAT terms, and the function ewp_set applies the function ε to a set of KAT terms.

```
Fixpoint ewp(t:kat)(a:atom) : bool :=
 match t with
 | kats x ⇒ false
 | katb b ⇒ evalT a b
 | katu t1 t2 ⇒ ewp t1 a || ewp t2 a
 | katc t1 t2 ⇒ ewp t1 a && ewp t2 a
 | katst t1 ⇒ true
 end.

Definition ewp_set(s:set kat)(a:atom) := fold (fun x ⇒ orb (ewp x a)) s false.

Fixpoint pdrv(x:kat)(a:atom)(s:sy) : set kat :=
 match x with
 | kats y ⇒ match _cmpA y s with
            | Eq ⇒ {katb ba1} | _ ⇒ ∅
            end
 | katb b ⇒ ∅
 | katu x1 x2 ⇒ pdrv x1 a s ∪ pdrv x2 a s
 | katc x1 x2 ⇒ if ewp x1 a then
                  dsr (pdrv x1 a s) x2 ∪ pdrv x2 a s
                else
                  dsr (pdrv x1 a s) x2
 | katst x1 ⇒ dsr (pdrv x1 a s) (katst x1)
 end.

Theorem pdrv_correct : ∀ a s r, SkatL (pdrv r a s) == LQ (kat2gl r) a s.

Theorem wpdrv_correct : ∀ w r, SkatL (wpdrv r w) == LQw (kat2gl r) w.
```

3.4 Finiteness of the Set of Partial Derivatives

Following Mirkin's notion of *pre-base* [Mir66] of a regular expressions, we now present a new way of determining the finiteness of the set of partial derivatives for any given KAT term. Kozen has presented a different notion of closure to prove the finiteness of the set of partial derivatives, but based on the sub-terms of a given KAT term.

Definition 3 *Let e be a* KAT *term. The* pre-base *of e, $\pi(e)$, is recursively defined by*

$$\begin{array}{lllllll} \pi(t) & = & \emptyset & \pi(e_1+e_2) & = & \pi(e_1) \cup \pi(e_2) \\ & & & \pi(e_1e_2) & = & \pi(e_1)e_2 \cup \pi(e_2) \\ \pi(p) & = & \{1\} & \pi(e^\star) & = & \pi(e)e^\star. \end{array} \tag{9}$$

The cardinality of $\pi(e)$ is bounded by the alphabetic size of e, that is, $\pi(e) \leq |e|_\Sigma$, where the alphabetic size $|e|_\Sigma$ is the number elements $p \in \Sigma$ in e. Let $\chi(e) = \{e\} \cup \pi(e)$. Thus, the cardinality of $\chi(e)$ is bounded by $|e|_\Sigma + 1$. The following theorem establishes that $\chi(e)$ contains the set of all derivatives of e and therefore we conclude that the set of all partial derivatives of any KAT term e is always finite.

Theorem 3 *Let e be a* KAT *term, and let $w \in (\mathsf{At} \cdot \Sigma)^\star$. Thus,*

$$\partial_{(\mathsf{At}\cdot\Sigma)^\star}(e) \subseteq \chi(e).$$

In the Coq development, the function π is encoded by the recursive function PI and χ by PD. The proof of Theorem 3 is given by theorem all_wpdrv_in_PD.

```
Fixpoint PI (e:kat) : set kat :=
 match e with
 | katb b ⇒ ∅
 | kats _ ⇒ {katb ba1}
 | katu x y ⇒ (PI x) ∪ (PI y)
 | katc x y ⇒ (dsr (PI x) y) ∪ (PI y)
 | katst x   ⇒ dsr (PI x) (katst x)
 end.

Definition PD(r:kat) := {r} ∪ (PI r).

Fixpoint sylen (e:kat) : nat :=
 match e with
 | kats _ ⇒ 1 | katb _ ⇒ 0
 | katu x y ⇒ sylen x + sylen y
 | katc x y ⇒ sylen x + sylen y
 | katst x ⇒ sylen x
 end.

Theorem PD_upper_bound : ∀ r, cardinal (PD r) ≤ (sylen r) + 1.

Theorem all_wpdrv_in_PD : ∀ w x r, x ∈ (wpdrv e w) → x ∈ PD(r).
```

4 A Procedure for Deciding KAT Term Equivalence

Given a KAT term e we know that

$$e \sim \mathsf{E}(e) \cup \left(\bigcup_{\alpha p \in (\mathsf{At}\cdot\Sigma)^\star} \alpha p \partial_{\alpha p}(e) \right), \tag{10}$$

and so, checking if $e_1 \sim e_2$ can be reformulated to checking the following two conditions:

$$\forall \alpha \in \mathsf{At},\ \varepsilon_\alpha(e_1) = \varepsilon_\alpha(e_2) \tag{11}$$

$$\forall \alpha p \in (\mathsf{At}\cdot\Sigma),\ \partial_{\alpha p}(e_1) \sim \partial_{\alpha p}(e_2) \tag{12}$$

This leads to an iterative procedure for deciding KAT terms equivalence by recursively testing the equivalence of sets of partial derivatives of e_1 and e_2.

Theorem 4 *Given* KAT *terms e_1 and e_2 defined over $\mathcal{B}$,Σ it holds that*

$$e_1 \sim e_2 \leftrightarrow \forall \alpha \in \mathsf{At}, \forall w \in (\mathsf{At}\cdot\Sigma)^\star, \varepsilon_\alpha(\partial_w(e_1)) = \varepsilon_\alpha(\partial_w(e_2)).$$

Corollary 1 *Let e_1 and e_2 be two* KAT *terms. If there exists an atom $\alpha \in \mathsf{At}$ and there exists a sequence $w \in (\mathsf{At} \cdot \Sigma)^\star$ such that*

$$\varepsilon_\alpha(\partial_w(e_1)) \neq \varepsilon_\alpha(\partial_w(e_2))$$

then it holds that $e_1 \not\sim e_2$.

The procedure EQUIVKAT, presented in Algorithm 1, specifies a computational interpretation of Theorem 4 and of Corollary 1. Given two KAT terms e_1 and e_2 this procedure corresponds to the iterated process of deciding the equivalence of their partial derivatives.

Algorithm 1 The procedure EQUIVKAT.

Require: $s = \{(\{e_1\}, \{e_2\})\}$, $h = \emptyset$
Ensure: true or false

```
procedure EquivKAT(s, h)
    while s ≠ ∅ do
        (Γ, Δ) ← POP(s)
        for α ∈ At do
            if ε_α(Γ) ≠ ε_α(Δ) then
                return false
            end if
        end for
        h ← h ∪ {(Γ, Δ)}
        for αp ∈ (At · Σ) do
            (Λ, Θ) ← ∂_αp(Γ, Δ)
            if (Λ, Θ) ∉ h then
                s ← s ∪ {(Λ, Θ)}
            end if
        end for
    end while
return true
end procedure
```

Two finite sets of derivatives are required to define EQUIVKAT: a set h that serves as an accumulator of derivatives already processed, and a set s that acts as a stack that gathers new derivatives yet to be processed. The set h ensures the termination of EQUIVKAT due to the finiteness of the number of derivatives and by ensuring that no derivative is considered in the algorithm more than once.

5 Implementation of EQUIVKAT in Coq

In this section we provide the details of the implementation of EQUIVKAT in the Coq proof assistant. This implementation follows along the lines of the implementation of the decision procedure for deciding regular expression equivalence presented in [MPdS11].

5.1 Pairs of KAT Derivatives

The pairs (Γ, Δ) in EQUIVKAT represent derivatives of the original KAT terms e_1 and e_2. This notion is captured by the *dependent record* type Drv presented below and whose fields are the actual pair of sets of KAT terms dp, a sequence w that is a member of $(\mathsf{At}\cdot\Sigma)^\star$, and a proof cw that witnesses that $dp = (\partial_w(\Gamma), \partial_w(\Delta))$, where the operator === stands for finite set equality.

```
Record Drv (e1 e2:kat) := mkDrv {
 dp :> set kat * set kat ;
 w : list AtSy ;
 cw : dp === (wpdrv w e1,wpdrv w e2)
}.
```

The definitions of derivation were extended to handle terms of type Drv, and are presented in the code below. The type AtSy is the type of pairs (p, α), such that $p \in \Sigma$ and $\alpha \in \mathsf{At}$.

```
Definition Drv_1st : Drv e1 e2 .
Proof.
 refine(Build_Drv ({e1},{e2}) ϵ _).
 abstract((* Proof that (∂ϵ({e1}),∂ϵ({e2})) = ({e1},{e2})*)).
Defined.

Definition Drv_pdrv (x:Drv e1 e2)(a:atom)(s:sy) : Drv e1 e2.
Proof.
 refine(match x with Build_ReW k w p ⇒ Build_Drv e1 e2 (pdrvp k a s) (w++((a,s)::ϵ)) _ end).
 abstract((* Proof that (∂wαp({e1}),∂ϵ({e2})) = ∂αp(∂w({e1}),∂w({e2})) *)).
Defined.

Definition Drv_wpdrv (w:list AtSy) : ReW e1 e2.
Proof.
 refine(Build_Drv e1 e2 (wpdrvp ({e1},{e2}) w) w _).
 abstract(reflexivity).
Defined.

Definition Drv_pdrv_set(s:Drv e1 e2)(sig:set AtSy) : set (Drv e1 e2) :=
 fold (fun x:AtSy ⇒ add (Drv_pdrv s (fst x) (snd x))) sig ∅.
```

5.2 Update of the Set of Derivatives

The body of the while-loop of EQUIVKAT's specification presented in Algorithm 1 is a sequence of two tasks: the first task consists on picking a pair (Γ, Δ) from the set s and checking if for all atoms $\alpha \in \mathsf{At}$ the equality $\varepsilon_\alpha(\Gamma) = \varepsilon_\alpha(\Delta)$ holds. The second task, that is executed only if the previous task succeeds, produces a new set of pairs s' such that

$$s' = (s \backslash \{(\Gamma, \Delta)\}) \cup \{\partial_{\alpha p}(\Gamma, \Delta) \mid \alpha p \in (\mathsf{At}\cdot\Sigma)\} \backslash (h \cup \{(\Gamma, \Delta)\}),$$

where $\partial_{\alpha p}(\Gamma, \Delta) = (\partial_{\alpha p}(\Gamma), \partial_{\alpha p}(\Delta))$.The function step implements the previous two tasks. It returns a term of type step_case whose constructors have the following reading: the constructor proceed indicates that a new set of derivatives was computed with success; the constructor termtrue indicates that there are no more pairs to be obtained from s and so h contains all the derivatives; finally, the constructor termfalse indicates that a pair (Γ, Δ) is a proof of in-equivalence.

```
Definition ewp_p(x:set kat * set kat)(a:atom) := eqb (ewp_set (fst x) a) (ewp_set (snd x) a).

Definition ewp_at_set(x:set kat * set kat)(ats:set atom) := fold (fun p ⇒ andb (ewp_p x p)) ats true

Definition ewpDrv(x:Drv e1 e2)(a:set atom) := ewp_at_set x a.
```

```
Definition newDrvSet(x:Drv e1 e2)(h:set (Drv e1 e2))(sig:set AtSy) : set (Drv e1 e2) :=
 filter (fun x ⇒ negb (x ∈ h)) (Drv_pdrv_set x sig).

Inductive step_case (e1 e2:kat) : Type :=
|proceed   : step_case e1 e2
|termtrue  : set (Drv e1 e2) → step_case e1 e2
|termfalse : Drv e1 e2 → step_case e1 e2.

Definition step(h s:set (Drv e1 e2))(sig:set sy)(ats:set atom) :
 ((set (Drv e1 e2) * set (Drv e1 e2)) * step_case e1 e2) :=
 match choose s with
 |None ⇒  ((h,s),termtrue e2 e1 h)
 |Some (de1,de2) ⇒
   if ewpDrv e1 e2 (de1,de2) ats then
    let h' := add (de1,de2) h in
     let rsd' := in
      let s' := newDrvSet e1 e2 (de1,de2) H' sig ats in
        (h',s' ∪ (s \ {(de1,de2)}),proceed e1 e2)
   else
    ((h,s),termfalse e1 e2 (de1,de2))
 end.
```

5.3 Encoding of EQUIVKAT

The function iterate implements the **while** loop of EQUIVKAT, takes two finite sets of terms of type Drv e_1 e_2, and returns a term of type term_cases whose constructors Equiv and NotEquiv indicate, respectively, the equivalence or the in-equivalence of the terms e_1 and e_2.

```
Inductive term_cases e1 e2 : Type :=
|Equiv : set (Drv e1 e2) → term_cases e1 e2
|NotEquiv : Drv e1 e2 → term_cases e1 e2.

Inductive DP (h s:set (Drv e1 e2))(ats:set atom) : Prop :=
|is_dp : h∩s === ∅ → ( ∀  x:atom, x ∈ ats) → ewpDrv_set e1 e2 h ats = true → DP h s ats.

Function iterate(e1 e2:kat)(h s:set (Drv e1 e2))(sig:set A)(d:DP e1 e2 h s)
 {wf (LLim e1 e2) h}: term_cases e1 e2 :=
  let ((h',s'),next) := step h s in
   match next with
    |termfalse x ⇒ NotEquiv e1 e2 x
    |termtrue  h ⇒ Equiv e1 e2 h
    |progress    ⇒  iterate e1 e2 h' s' sig (DP_upd e1 e2 h s sig D)
  end.
Proof.
 (* Proof obligation 1 : proof that LLim is a decreasing measure for iterate *)
 abstract(apply DP_wf).
 (* Proof obligation 2: proof that LLim is a well founded relation. *)
 exact(guard e1 e2 100 (LLim_wf e1 e2)).
Defined.
```

We have used the `Function` command [BC02] that helps users in defining non *structurally decreasing* recursive function within Coq's type theory. The decoration { wf (LLim e_1 e_2) } has the purpose of informing the inner mechanism of Function that the recursive definition must follow the *well-founded relation* LLim. This relation relates two sets h and h', such that

$$\mathsf{LLim}\, e_1\, e_2\, (h, h') = T - |h'| < T - |h|,$$

where $T = (2^{(|e_1|_\Sigma+1)} \times 2^{(|e_2|_\Sigma+1)} + 1)$, that is, the set containing all the possible combinations of the derivatives of e_1 and e_2. The proof that LLim is well founded corresponds to a checkable evidence of the termination of iterate and it is used as input to the guard function in order to discharge the second proof obligation produced by the Function command. The purpose of the function guard is to avoid that LLim_wf is explicitly computed by Coq's reduction mechanisms, which leads to highly inefficient computation times [1].

The last argument of iterate is a term d of the dependent type DP. This type contains a proof that the sets s and h are always disjoint, a proof that all the pairs (Γ, Δ) in h represent equivalent languages, and a proof that all the atoms are members of the set ats. Note also that s and h being always disjoint along the execution of iterate ensures that the set h increases in each recursive call and thus satisfies the well founded relation LLim.

The function equivkat_aux lifts the result of iterate into its Boolean counterpart. The function equivkat fully implements EQUIVKAT and is simply a call to equivkat_aux with the correct values of s and h as specified in Algorithm 1.

```
Definition equivkat_aux(e1 e2:kat)(h s:set (Drv e1 e2))(sig:set sy)(d:DP e1 e2 h s):=
 let h' := iterate e1 e2 h s sig D in
  match h' with
  | Ok _      ⇒ true
  | NotOk _ ⇒ false
 end.

Definition mkDP_1st : DP e1 e2 ∅ {Drv_1st e1 e2}.

Definition equivkat(e1 e2:kat) := equivkat_aux e1 e2 ∅ {Drv_1st e1 e2} sigmaP (mkDP_1st e1 e2).
```

5.4 Correctness of equivkat

The correctness of equivkat consists on proving that: (1) whenever equivkat e_1 e_2 returns true then it implies Theorem 4, which directly leads to KAT term equivalence; (2) whenever equivkat e_1 e_2 returns false then a derivative (Γ, Δ) exists such that $\varepsilon_\alpha(\Gamma) \neq \varepsilon_\alpha(\Delta)$, which in turn implies $e_1 \not\sim e_2$ since $\alpha \in \mathsf{G}(\Gamma)$ and $\alpha \notin \mathsf{G}(\Delta)$, as stated in Corollary 1.

In order to prove (1) we follow the approach described in [MPdS11], where an *invariant* is defined over iterate which states that in each recursive call all the derivatives (Γ, Δ) that belong to the accumulator set h have all of their derivatives either in h already, or are in the set s. This invariant is given by the inv_iterate predicate presented below. The auxiliary lemma invP_iterate_ind_correct provides the evidence that if iterate terminates and returns a term Equiv e_1 e_2 x, where x is the set of all derivatives of e_1 and e_2. Lemma invP_iterate_eq_gl proves that iterate leads to language equivalence and is used to prove the main lemma equivkat_true_correct.

```
Definition invP(h s:set (Drv e1 e2))(ats:set atom)(sig:set sy) :=
  ∀ x, x ∈ h → ∀ a, a ∈ sig → ∀ b, b ∈ ats → (Drv_pdrv e1 e2 x b a) ∈ (h ∪ s).

Definition invP_iterate(h s:set (Drv e1 e2))(ats:set atom)(sig:set sy) :=
 (Drv_1st e1 e2) ∈ (h ∪ s) ∧ ( ∀ x, x ∈ (h ∪ s) → ewp_Drv e1 e2 x ats = true) ∧ invP h s
ats sig.

Lemma invP_iterate_ind_correct' : ∀ h s ats sig d x,
```

[1] The usage of guard was proposed by Bruno Barras and improved by Georges Gonthier in the Coq-club mailing list and has been used in other works that require computation of functions defined in Coq that involve well-founded relations.

```
invP h s ats sig → iterate e1 e2 h s ats sig d = Equiv e1 e2 x → invP x ∅ ats sig.

Lemma invP_iterate_eq_gl : ∀ x ats,
 iterate e1 e2 ∅ {Drv_1st e1 e2} ats sigmaP (mkDP_ini e1 e2 ats) = Equiv e1 e2 x →
        invP_iterate e1 e2 x ∅ ats sigmaP → (kat2gl e1) == (kat2gl e2).

Theorem equivkat_true_correct :
  equivkat e1 e2 ats sigmaP = true → (kat2gl e1) == (kat2gl e2).
```

The proof of (2) is also carried out by induction over iterate, but there is no need to establish any sort of invariant. We obtain the desired results by performing case analysis over the value returned by step: if it returns a the term NotEquiv e_1 e_2 x, where $x = (\Gamma, \Delta)$ then the inequality $\varepsilon_\alpha(\Gamma) \neq \varepsilon_\alpha(\Delta)$ must hold. By Corollary 1 this leads to $\alpha \in \mathsf{G}(\Gamma)$ and $\alpha \notin \mathsf{G}(\Delta)$, or vice versa, that is, $e_1 \not\sim e_2$. This logical condition is given by the lemmas iterate_false and iterate_false_correct, and by the theorem equivkat_false_correct presented below.

```
Lemma iterate_false : ∀ h s ats sig d x,
 iterate e1 e2 h s ats sig d = NotEquiv e1 e2 x → ewp_Drv e1 e2 x ats = false.

Lemma correct_aux_2 : ∀ s ats sig,
 iterate e1 e2 ∅ {Drv_1st e1 e2} ats sig (mkDP_ini e1 e2 ats) = NotEquiv e1 e2 s →
    equivkat e1 e2 = false.

Theorem equivkat_false_correct : equivkat e1 e2 = false → ¬((kat2gl r1) == (kat2gl r2)).
```

6 Application to Program Verification

The main motivation behind the implementation of equivkat is to provide a certified decision procedure that can be used to help on the construction of partial correctness proofs over simple imperative programs. As an example let us consider the program Fact that computes the factorial of a non-negative integer x. This example was obtained from [ABM12].

In order to transform to KAT we need Fact to be fully annotated, and we have to eliminate the assignments. In the table below we present the encoding of the Hoare triple $\{true\}$Fact$\{y = x!\}$ in KAT, where we associate to each assertion a test t_i, and to each assignment a program p_i.

Fact	Encoding
{true}	t_0
y := 1	p_1
$\{y = 0!\}$	t_1
z := 0 ;	p_2
$\{y = z!\}$	t_2
while ¬(z=x) do	t_3
{	
$\{y = z!\}$	t_2
z := z + 1 ;	p_3
$\{y \times z = z!\}$	t_4
y := y * z ;	p_4
}	
$\{y = x!\}$	t_5

The final encoding in KAT is the equality

$$t_0p_1t_1p_2t_2(t_3t_2p_3t_4p_4)^\star\overline{t_3t_5} = 0. \tag{13}$$

To prove (13) we need an extra set of hypoteses that can be obtained in a backward fashion [ABM12]. These hypotheses are of the form $r_i = 0$ and correspond to Hoare triples. Thus, to prove equation (13) we need to prove a KAT implication of the form

$$r_0 = 0 \wedge r_1 = 0 \wedge \ldots \wedge r_k = 0 \rightarrow e_1 = e_2$$

where $e_1 = e_2$ is equation (13). Kozen showed in [Koz00] that the validity of previous implication is tantamount to the validity of the equality $e_1 + uru = e_2 + uru$, such that $u = (p_0 + \ldots + p_n)^\star$ with $\Sigma = \{p_0, \ldots, p_n\}$ and $r = r_0 + \ldots r_k$. For the case of Fact we have:

- $u = (p_1 + p_2 + p_3 + p_4)^\star$
- $r = t_0p_1\overline{t_1} + t_1p_2\overline{t_2} + t_3t_2p_3\overline{t_4} + t_4p_2\overline{t_2} + t_2\overline{t_3t_5}$

Our decision procedure proved the validity of the equation

$$t_0p_1t_1p_2t_2(t_3t_2p_3t_4p_4)^\star\overline{t_3t_5} + uru = 0 + uru$$

and so the program Fact is correct. The time needed to perform the proof was 22 seconds. The procedure is not very efficient due to the cost of calculating the function ε and the cost of the derivation for each pair (Γ, Δ) considered during the execution of the decision procedure. Nevertheless, the procedure can be extracted as a functional program that can be compiled outside Coq in order to obtain faster computations.

7 Conclusions

In this paper we have presented the mechanization of a decision procedure for KAT terms. The overall development includes the formalization of the language-theoretic model of sets of guarded strings and a new proof of the finiteness of the set of partial derivatives. The Coq code for the whole development is available in [MPM].

We have showed that our procedure can be used to automatically prove the partial correctness of simple imperative programs, encoded in PHL. This encoding can be automated by applying one of the standard Verification Condition Generator available and a translator that associates assignments to primitive programs, and assertions to tests.

The procedure is not yet very efficient due to the way we handle the Boolean part of KAT. Currently, we are investigating ways to use SAT solvers inside of Coq. Moreover, we feel that it is important to investigate how to generate the set of all atoms At, possibly in a lazy way and without resorting on the totality of the $2^{|\mathcal{B}|}$ elements of At.

References

[ABM12] Ricardo Almeida, Sabine Broda, and Nelma Moreira. Deciding KAT and Hoare logic with derivatives. Submitted, 2012.

[AHK06] Kamal Aboul-Hosn and Dexter Kozen. KAT-ML: An interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1–2):9–33, 2006.

[AK01] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Cornell University, 2001.

[BC02] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In Victor Carreño, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 31–46. Springer, 2002.

[BP10] Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, October 1964.

[CS] Thierry Coquand and Vincent Siles. A decision procedure for regular expression equivalence in type theory. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP 2011, Kenting, Taiwan, December 7-9, 2011.*, number 7086 in LNCS, pages 119–134. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HS07] Peter Höfner and Georg Struth. Automated reasoning in Kleene algebra. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2007.

[Kle] S. Kleene. *Representation of events in nerve nets and finite automata*, pages 3–42. Princeton University Press, shannon, C. and McCarthy, J. edition.

[KN11] Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, 2011. Published online.

[Kom] Vladimir Komendantsky. Computable partial derivatives of regular expressions. `http://www.cs.st-andrews.ac.uk/~vk/papers.html`.

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[Koz00] Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000.

[Koz01] Dexter Kozen. Automata on guarded strings and applications. Technical report, Cornell University, Ithaca, NY, USA, 2001.

[Koz08] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.

[KS96] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, pages 244–259, 1996.

[KT01] Dexter Kozen and Jerzy Tiuryn. On the completeness of propositional Hoare logic. *Inf. Sci.*, 139(3-4):187–195, 2001.

[McC] William McCune. Prover9 and Mace4. http://www.cs.unm.edu/smccune/mace4. Access date: 1.10.2011.

[Mir66] B.G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.

[MP08] Nelma Moreira and David Pereira. KAT and PHL in Coq. *CSIS*, 05(02), December 2008. ISSN: 1820-0214.

[MPdS11] Nelma Moreira, David Pereira, and Simao Melo de Sousa. Deciding regular expression (in-)equivalence in Coq. Technical Report DCC-2011-06, DCC-FC & LIACC, Universidade do Porto, 2011.

[MPM] Nelma Moreira, David Pereira, and Simão Melo de Sousa. Source code of the formalization. http://www.liacc.up.pt/~kat/equivKAT.tgz.

[The] The Coq Development Team. http://coq.inria.fr.