

Formalization of Predictive Runtime Enforcement

...

August 27, 2015

Contents

1	Main results	1
2	Example	9
theory <i>Predictive</i> imports <i>Main</i> begin		

1 Main results

In this formalization we use lists of some type variable "'a" to model words over "'a"

Syntax for the lattice operations:

notation
bot (\perp) **and**
top (\top) **and**
inf (**infixl** \sqcap 70) **and**
sup (**infixl** \sqcup 65)

We introduce the prefix relation on lists as an instantiation of a partial order relation. The fact that x is a prefix of a list y is denoted by $x \leq y$. We prove that the prefix relation is a partial order, and we also prove some additional properties

instantiation *list* :: (*type*) *order* **begin**
primrec *less-eq-list* **where**
($[] \leq x$) = *True* |
($(a \# x) \leq y$) = (*case y of* $[] \Rightarrow$ *False* | $b \# z \Rightarrow a = b \wedge (x \leq z)$)

definition *less-list-def*: $((x::'a \text{ list}) < y) = (x \leq y \wedge \neg y \leq x)$

lemma [*simp*]: $(x::'a \text{ list}) \leq x$
by (*induction x, simp-all*)

lemma *prefix-antisym*: $\bigwedge y. (x::'a \text{ list}) \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
apply (*induction x*)

```

apply (case-tac y, simp-all)
by (case-tac y, simp-all)

lemma prefix-trans:  $\bigwedge y z . (x :: 'a \text{ list}) \leq y \implies y \leq z \implies x \leq z$ 
apply (induction x, simp-all)
apply (case-tac y, simp-all)
by (case-tac z, simp-all, auto)

lemma [simp]:  $(y \leq []) = (y = [])$ 
by (unfold less-eq-list-def, case-tac y, auto)

lemma [simp]:  $[] \leq ax$ 
by (simp add: less-eq-list-def)

lemma prefix-concat:  $\bigwedge x . x \leq y = (\exists z . y = x @ z)$ 
apply (induction y, simp-all)
by (case-tac x, auto)

lemma [simp]:  $(\text{butlast } x) \leq x$ 
by (induction x, simp-all)

lemma prefix-butlast:  $\bigwedge y . (x \leq y) = (x = y \vee x \leq (\text{butlast } y))$ 
proof (induction x)
  case Nil show ?case by simp
  case (Cons x xs)
    assume A:  $\bigwedge y . (xs \leq y) = (xs = y \vee xs \leq (\text{butlast } y))$ 
    show ?case
      apply simp
      apply (case-tac y, simp)
      apply simp
      apply safe
      apply simp-all
      apply (subst (asm) A, simp)
      by (subst A, simp)
qed

lemma [simp]:  $(x @ y \leq x) = (y = [])$ 
by (induction x, simp-all)

instance proof
  qed (simp-all add: less-list-def prefix-antisym, rule prefix-trans)
end

```

A finite deterministic automaton is modeled as a record of a transition function $\delta : 's \rightarrow 'a \rightarrow 's$ and a set $Final : 's \text{ set}$ of final states, and an initial state $s_0 : 's$. The states of the automaton are from the type variable $'s$, and the letters of the alphabet from $'a$.

```

record ('s, 'a) automaton =
   $\delta :: 's \Rightarrow 'a \Rightarrow 's$ 

```

$Final :: 's \text{ set}$
 $s_0 :: 's$

The language of an automaton A is a predicate on lists of letters, and it is defined by primitive recursion:

primrec $lang :: ('s, 'a, 'c) \text{ automaton-ext} \Rightarrow 'a \text{ list} \Rightarrow bool$ **where**
 $lang\ A\ [] = ((s_0\ A) \in (Final\ A)) \mid$
 $lang\ A\ (a \# x) = lang\ (A[s_0 := \delta\ A\ (s_0\ A)\ a])\ x$

We extend the transition function δ from letters to lists of letters, also by primitive induction

primrec $\delta e :: ('s, 'a, 'b) \text{ automaton-ext} \Rightarrow 'a \text{ list} \Rightarrow 's$ **where**
 $\delta e\ A\ [] = s_0\ A \mid$
 $\delta e\ A\ (a \# x) = \delta e\ (A[s_0 := \delta\ A\ (s_0\ A)\ a])\ x$

Next two lemma connect the language definition to the extended transition function.

lemma $lang\-\delta e a e :: \bigwedge A . lang\ A\ x = ((\delta e\ A\ x) \in Final\ A)$
by ($induction\ x, simp\text{-}all$)

lemma $lang\-\delta e a e b :: \bigwedge y\ A . lang\ A\ (x @ y) = lang\ (A[s_0 := (\delta e\ A\ x)])\ y$
by ($induction\ x, simp\text{-}all$)

We introduce the standard product construction of two automata. Here we construct the product corresponding the intersection of the languages of the two automata.

definition $product :: ('s, 'a, 'c) \text{ automaton-ext} \Rightarrow ('t, 'a, 'c) \text{ automaton-ext}$
 $\Rightarrow ('s \times 't, 'a, 'c) \text{ automaton-ext}$ (**infix** $**\ 60$) **where**
 $A ** B = ()$
 $\delta = (\lambda (s, t)\ a . (\delta\ A\ s\ a, \delta\ B\ t\ a)),$
 $Final = Final\ A \times Final\ B,$
 $s_0 = (s_0\ A, s_0\ B),$
 $\dots = automaton.\text{more}\ A()$

Next five lemmas are straightforward properties of the product of two automata.

lemma [$simp$]: $s_0\ (A ** B) = (s_0\ A, s_0\ B)$
by ($simp\ add: product\text{-}def$)

lemma [$simp$]: $Final\ (A ** B) = (Final\ A \times Final\ B)$
by ($simp\ add: product\text{-}def$)

lemma [$simp$]: $(A ** B)[s_0 := (s, t)] = (A[s_0 := s]) ** (B[s_0 := t])$
by ($simp\ add: product\text{-}def$)

lemma [$simp$]: $\delta\ (A ** B)\ (s, t)\ a = (\delta\ A\ s\ a, \delta\ B\ t\ a)$
by ($simp\ add: product\text{-}def$)

lemma [simp]: $\bigwedge A B . \delta e (A ** B) x = (\delta e A x, \delta e B x)$
by (induction x, simp-all)

Next two lemmas show that the language of the product is the intersection of the languages of the automata. Second lemma is the point-free version of the first lemma.

lemma intersection-aux: $\bigwedge A B . \text{lang } (A ** B) x = (\text{lang } A x \wedge \text{lang } B x)$
apply (induction x)
by (auto simp add: lang-deltae)

lemma intersection: $\text{lang } (A ** B) = (\text{lang } A \sqcap \text{lang } B)$
by (simp add: fun-eq-iff intersection-aux)

Next declaration introduces the complement of an automaton as a instantiation of the Isabelle uminus class. We take this approach because we what to use the unary symbol $-$ for the complement. Otherwise this is just a simple definition similar to the product.

instantiation automaton-ext :: (type, type, type) uminus **begin**
definition complement-def: $- A = A(| \text{Final} := - \text{Final } A |)$
instance proof qed
end

Next five lemmas give some properties of the complement.

lemma [simp]: $\delta (-A) = \delta A$
by (simp add: complement-def)

lemma [simp]: $s_0 (-A) = s_0 A$
by (simp add: complement-def)

lemma complement-init[simp]: $(-A)(| s_0 := s |) = -(A(| s_0 := s |))$
by (simp add: complement-def)

lemma [simp]: $\bigwedge A . \delta e (-A) x = \delta e A x$
by (induction x, simp-all)

lemma [simp]: $\text{Final } (-A) = - \text{Final } A$
by (simp add: complement-def)

The language of the complement of A is the complement of the language of A . Next two lemmas express this property in point-wise and point-free manner.

lemma complement-aux: $\bigwedge A . \text{lang } (- A) x = (\neg (\text{lang } A x))$
by (simp add: lang-deltae)

lemma complement: $\text{lang } (-A) = (\neg (\text{lang } A))$
by (simp add: fun-eq-iff complement-aux)

Next definition introduces an automaton $Prefix\ A$ based on automaton A . A list x is in the language of $Prefix\ A$ if and only if there is a prefix of x in the language of A . In the paper this automaton is denoted by B_φ , where $\varphi = lang\ A$.

definition $Prefix\ A = ()$
 $\delta = (\lambda\ u\ a\ .\ (case\ u\ of\ None \Rightarrow None\ |\ Some\ s \Rightarrow if\ s \in Final\ A\ then\ None$
 $else\ Some\ (\delta\ A\ s\ a))),$
 $Final = Some\ ' (Final\ A) \cup \{None\},$
 $s_0 = Some\ (s_0\ A),$
 $\dots = more\ A()$

lemma $[simp]: s_0\ (Prefix\ A) = Some\ (s_0\ A)$
by $(simp\ add: Prefix-def)$

We introduce some properties of $Prefix\ A$ in the next six lemmas.

lemma $Prefix-initial[simp]: Prefix\ A()s_0 := Some\ s = Prefix\ (A()s_0 := s)$
apply $(auto\ simp\ add: Prefix-def\ fun-eq-iff)$
by $(case-tac\ x,\ simp-all)$

lemma $[simp]: lang\ ((Prefix\ A)()s_0 := None) x$
by $(induction\ x,\ simp-all\ add: Prefix-def)$

lemma $[simp]: s \in Final\ A \implies \delta\ (Prefix\ A)\ (Some\ s)\ a = None$
by $(simp\ add: Prefix-def)$

lemma $[simp]: s \notin Final\ A \implies \delta\ (Prefix\ A)\ (Some\ s)\ a = Some\ (\delta\ A\ s\ a)$
by $(simp\ add: Prefix-def)$

lemma $[simp]: s \in Final\ A \implies lang\ ((Prefix\ A)()s_0 := Some\ s) x$
by $(case-tac\ x,\ simp-all,\ simp\ add: Prefix-def)$

lemma $lang\ ((Prefix\ A)()s_0 := Some\ s) = \top \implies s \in Final\ A$
apply $(simp\ add: fun-eq-iff\ Prefix-def\ image-def)$
by $(drule-tac\ x = []\ in\ spec,\ simp)$

The language of $Prefix\ A$ in terms of the language of A is given by the next lemma. This is Lemma 5 from the paper [1].

lemma $Prefix-lang: \bigwedge (A::('s, 'a, 'c)\ automaton-ext) . lang\ (Prefix\ A)\ x = (\exists\ y$
 $. lang\ A\ y \wedge y \leq x)$
proof $(induction\ x)$
case (Nil) **show** $?case$
by $(simp\ add: Prefix-def\ image-def)$
next
case $(Cons\ a\ x)$
assume $A: \bigwedge (A::('s, 'a, 'c)\ automaton-ext) . lang\ (Prefix\ A)\ x = (\exists\ y. lang$
 $A\ y \wedge y \leq x)$
from A **have** $B: \bigwedge (A::('s, 'a, 'c)\ automaton-ext) y . lang\ A\ y \implies y \leq x$
 $\implies lang\ (Prefix\ A)\ x$

```

    by blast
  show ?case
  apply simp
  apply safe
  apply (case-tac s0 A ∈ Final A, simp-all)
  apply (rule-tac x = [] in exI, simp)
  apply (simp add: A, safe)
  apply (rule-tac x = a # y in exI)

  apply simp
  apply (case-tac y, simp-all, safe)
  apply (case-tac s0 A ∈ Final A, simp-all)
  by (drule B, simp-all)
qed

```

Next definition introduces $k_{\psi, \varphi}$ function from Definition 2 in the paper [1]. The automata A_ψ and A_φ correspond to the properties ψ and φ , respectively.

definition $kfunc\ A_\psi\ A_\varphi\ x = (\forall\ y.\ lang\ A_\psi\ (x @ y) \longrightarrow (\exists\ z.\ (lang\ A_\varphi\ (x @ z)) \wedge z \leq y))$

The Urgency constraint is introduced in the next definition, and it has as hypothesis the $kfunc$ function.

definition $Urgency\ A_\psi\ A_\varphi\ Enf = (\forall\ x.\ kfunc\ A_\psi\ A_\varphi\ x \longrightarrow Enf\ x = x)$

The weaker version of Urgency is introduced by:

definition $Urgency'\ A_\psi\ A_\varphi\ Enf = (\forall\ x.\ (\forall\ y.\ lang\ A_\psi\ (x @ y) \longrightarrow lang\ A_\varphi\ x) \longrightarrow Enf\ x = x)$

lemma $Urgency\text{-}Urgency'\text{-}aux: (\forall\ y.\ lang\ A_\psi\ (x @ y) \longrightarrow lang\ A_\varphi\ x) \implies kfunc\ A_\psi\ A_\varphi\ x$

by (metis $kfunc\text{-}def\ append\text{-}Nil2\ less\text{-}eq\text{-}list.simps(1)$)

Urgency is stronger than Urgency' (Lemma 2 in [1]):

lemma $Urgency\text{-}Urgency': Urgency\ A_\psi\ A_\varphi\ Enf \implies Urgency'\ A_\psi\ A_\varphi\ Enf$

by (simp add: $Urgency'\text{-}def\ Urgency\text{-}def\ Urgency\text{-}Urgency'\text{-}aux$)

When the property ψ is true for all sequences, then the Urgency property is simplified to the non-predictive case (Lemma 3 in [1]).

lemma $no\text{-}prediction: lang\ A_\psi = \top \implies Urgency\ A_\psi\ A_\varphi\ Enf = (\forall\ x.\ lang\ A_\varphi\ x \longrightarrow Enf\ x = x)$

apply (auto simp add: $Urgency\text{-}def\ kfunc\text{-}def$)

apply (metis $append\text{-}Nil2\ less\text{-}eq\text{-}list.simps(1)$)

by (metis $list.simps(4)\ neg\text{-}Nil\text{-}conv\ less\text{-}eq\text{-}list.simps(2)\ self\text{-}append\text{-}conv$)

Next definition is a more abstract variant of $kfunc$ as an inclusion of regular languages. Here we do not have the existential quantifier.

definition $kfunc\text{-}lang\ A_\psi\ A_\varphi\ x = (lang\ (A_\psi \parallel s_0 := (\delta e\ A_\psi\ x))) \leq lang\ ((Prefix\ A_\varphi) \parallel s_0 := Some\ (\delta e\ A_\varphi\ x)))$

lemma *kfunc-kfunc-lang*: $kfunc\ A_\psi\ A_\varphi\ x = kfunc\text{-}lang\ A_\psi\ A_\varphi\ x$
by (*simp add: kfunc-def kfunc-lang-def lang-deltaeb Predictive.Prefix-lang le-fun-def*)

lemma *kfunc-lang-empty*: $kfunc\text{-}lang\ A_\psi\ A_\varphi\ x = (lang\ ((A_\psi\ ** - (Prefix\ A_\varphi)))(s_0 := (\delta e\ A_\psi\ x, Some\ (\delta e\ A_\varphi\ x)))) = \perp$
by (*simp add: intersection Predictive.complement kfunc-lang-def fun-eq-iff le-fun-def*)

Next theorem shows the implementation of *kfunc* as a test of emptiness of a regular language (Theorem 2 in [1]).

theorem *kfunc-empty*: $kfunc\ A_\psi\ A_\varphi\ x = (lang\ ((A_\psi\ ** - (Prefix\ A_\varphi)))(s_0 := (\delta e\ A_\psi\ x, Some\ (\delta e\ A_\varphi\ x)))) = \perp$
by (*unfold kfunc-kfunc-lang kfunc-lang-empty, simp*)

Next definition introduces the enforcement function. In this formalization we chose to define *enforce* directly while in [1] is defined using another function called *store* that returns two sequences. The *enforce* function is the first component of *store*.

fun *enforce* :: $('s, 'a, 'c)\ automaton\text{-}ext \Rightarrow ('t, 'a, 'c)\ automaton\text{-}ext \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
enforce $A_\psi\ A_\varphi\ x =$
 (*if* $x = []$ *then*
 $[]$
else
 (*if* $kfunc\ A_\psi\ A_\varphi\ x$ *then*
 x
else
 $enforce\ A_\psi\ A_\varphi\ (butlast\ x))$)

Next three lemmas are used in the proofs of soundness, transparency, and urgency. These lemmas correspond to Lemma 4 from [1].

lemma *kfunc-enforce*: $enforce\ A_\psi\ A_\varphi\ x \neq [] \Longrightarrow kfunc\ A_\psi\ A_\varphi\ (enforce\ A_\psi\ A_\varphi\ x)$

apply (*induction x rule: length-induct*)
apply (*subst enforce.simps*)
apply (*case-tac xs = []*)
apply *simp*
apply (*simp del: enforce.simps, safe*)
apply (*drule-tac x = butlast xs in spec, safe*)
apply (*simp-all del: enforce.simps*)
by *simp*

lemma *kfunc-prefix-enforce*: $kfunc\ A_\psi\ A_\varphi\ y \Longrightarrow y \leq x \Longrightarrow y \leq (enforce\ A_\psi\ A_\varphi\ x)$

apply (*induction x rule: length-induct*)
apply (*subst enforce.simps*)
apply (*case-tac xs = []*)

```

apply simp
apply (simp del: enforce.simps, safe)
apply (drule-tac x = butlast xs in spec, safe)
apply (simp-all del: enforce.simps)
by (subst (asm) prefix-butlast, simp)

```

```

lemma lang-enf-kfunc: lang  $A_\varphi$  x  $\implies$  kfunc  $A_\psi$   $A_\varphi$  x
apply (simp add: kfunc-def, safe)
by (rule-tac x = [] in exI, simp)

```

Finally we prove the enforcement function satisfies soundness, transparency, and urgency properties.

```

theorem Transparency1: enforce  $A_\psi$   $A_\varphi$  x  $\leq$  x
apply (induction x rule: length-induct)
apply (subst enforce.simps)
apply (case-tac xs = [])
apply (unfold if-P)
apply simp
apply (unfold if-not-P)
apply (case-tac kfunc  $A_\psi$   $A_\varphi$  xs)
apply simp
apply (unfold if-not-P)
apply (drule-tac x = butlast xs in spec)
apply safe
apply simp
by (rule-tac y = butlast xs in prefix-trans, simp-all)

```

```

theorem Transparency2: lang  $A_\varphi$  x  $\implies$  enforce  $A_\psi$   $A_\varphi$  x = x
by (simp add: lang-enf-kfunc)

```

```

theorem Urgency: kfunc  $A_\psi$   $A_\varphi$  x  $\implies$  enforce  $A_\psi$   $A_\varphi$  x = x
by simp

```

```

theorem Soundness: lang  $A_\psi$  x  $\implies$  enforce  $A_\psi$   $A_\varphi$  x  $\neq$  []  $\implies$  lang  $A_\varphi$  (enforce
 $A_\psi$   $A_\varphi$  x)

```

proof –

```

assume A: lang  $A_\psi$  x
assume B: enforce  $A_\psi$   $A_\varphi$  x  $\neq$  []
have enforce  $A_\psi$   $A_\varphi$  x  $\leq$  x by (rule Transparency1)
from this obtain z where D: x = enforce  $A_\psi$   $A_\varphi$  x @ z by (simp add:
prefix-concat del: enforce.simps, safe, simp)
from A and this have [simp]: lang  $A_\psi$  (enforce  $A_\psi$   $A_\varphi$  x @ z) by simp
from B have kfunc  $A_\psi$   $A_\varphi$  (enforce  $A_\psi$   $A_\varphi$  x) by (rule kfunc-enforce)
from this have C:  $\bigwedge y . \text{lang } A_\psi (\text{enforce } A_\psi A_\varphi x @ y) \implies (\exists t . \text{lang } A_\varphi$ 
(enforce  $A_\psi$   $A_\varphi$  x @ t)  $\wedge t \leq y)$  by (simp add: kfunc-def)
have ( $\exists t . \text{lang } A_\varphi (\text{enforce } A_\psi A_\varphi x @ t) \wedge (t \leq z)$ ) by (rule C, simp
del: enforce.simps)
then obtain za where F: lang  $A_\varphi$  (enforce  $A_\psi$   $A_\varphi$  x @ za) and E: za  $\leq$  z

```



```

by blast
  from this have kfunc  $A_\psi$   $A_\varphi$  (enforce  $A_\psi$   $A_\varphi$   $x$  @  $za$ ) by (simp add:
lang-enf-kfunc del: enforce.simps)
  from this have enforce  $A_\psi$   $A_\varphi$   $x$  @  $za \leq$  enforce  $A_\psi$   $A_\varphi$   $x$  apply (rule
kfunc-prefix-enforce)
  by (cut-tac  $D$   $E$ , simp add: prefix-concat del: enforce.simps, blast)
  from this have [simp]:  $za = []$  by simp
  from  $F$  show ?thesis by (simp del: enforce.simps)
qed

```

2 Example

```

datatype  $Sa = l0 \mid l1 \mid l2$ 
datatype  $Sig = a \mid b \mid c$ 
datatype  $Sb = k0 \mid k1 \mid k2 \mid k3$ 

```

```

fun
   $\delta a :: Sa \Rightarrow Sig \Rightarrow Sa$ 
where
   $\delta a \ l0 \ a = l0 \mid$ 
   $\delta a \ l0 \ b = l1 \mid$ 
   $\delta a \ l1 \ c = l0 \mid$ 
   $\delta a \ - \ a = l2 \mid$ 
   $\delta a \ - \ b = l2 \mid$ 
   $\delta a \ - \ c = l2$ 

```

```

definition  $Fa = \{l0\}$ 

```

```

fun
   $\delta b :: Sb \Rightarrow Sig \Rightarrow Sb$ 
where
   $\delta b \ k0 \ a = k0 \mid$ 
   $\delta b \ k0 \ b = k1 \mid$ 
   $\delta b \ k1 \ a = k0 \mid$ 
   $\delta b \ k1 \ c = k2 \mid$ 
   $\delta b \ k2 \ a = k0 \mid$ 
   $\delta b \ - \ a = k3 \mid$ 
   $\delta b \ - \ b = k3 \mid$ 
   $\delta b \ - \ c = k3$ 

```

```

definition  $Fb = \{k0, k1, k2\}$ 

```

```

lemma kfunc-lang ( $\delta = \delta b$ ,  $Final = Fb$ ,  $s_0 = k0$ ) ( $\delta = \delta a$ ,  $Final = Fa$ ,  $s_0 =$ 
 $l0$ ) [ $a, b$ ] = False
  apply (simp add: kfunc-lang-def le-fun-def)
  apply (rule-tac  $x = []$  in exI)
  by (simp add: Fb-def Prefix-def Fa-def )

```

```

lemma kfunc ( $\delta = \delta b$ ,  $Final = Fb$ ,  $s_0 = k0$ ) ( $\delta = \delta a$ ,  $Final = Fa$ ,  $s_0 = l0$ ) [ $a$ ]

```

```
    apply (simp add: kfunc-def Fb-def Fa-def, auto)  
    by (rule-tac x = [] in exI, simp)  
end
```

References

- [1] ... Predictive runtime enforcement. Sept. 2015. to appear.