Formalization of Predictive Runtime Enforcement

...

August 28, 2015

Contents

theory *Predictive* **imports** *Main* **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 ($\bot$) **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**
$([] \leq x) = True$ |
$((a \# x) \leq y) = (case\ y\ of\ [] \Rightarrow False \mid b \# z \Rightarrow a = b \wedge (x \leq z))$

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

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

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

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

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

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

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

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

lemma [*simp*]: $(butlast\ x) \le x$
by (*induction* x, *simp-all*)

lemma *prefix-butlast*: $\bigwedge\ y\ .\ (x \le y) = (x = y \lor x \le (butlast\ y))$
proof (*induction* x)
case *Nil* **show** *?case* **by** *simp*
case (*Cons x xs*)
assume A: $\bigwedge\ y\ .\ (xs \le y) = (xs = y \lor xs \le (butlast\ y))$
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*]: $(x\ @\ y \le x) = (y = [])$
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 $\delta :' s \to' a \to' s$ and a set $Final :' sset$ 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* $=$
$\delta :: 's \Rightarrow 'a \Rightarrow 's$

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

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

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

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

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

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

lemma *lang-deltae*: $\bigwedge\ A\ .\ lang\ A\ x = ((\delta e\ A\ x) \in Final\ A)$
by (*induction x*, *simp-all*)

lemma *lang-deltaeb*: $\bigwedge\ y\ A\ .\ lang\ A\ (x\ @\ y) = lang\ (A(\!|s_0{:=}\ (\delta e\ A\ x)|\!))\ y$
by (*induction x*, *simp-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 $product :: ('s,\ 'a,\ 'c)\ automaton\text{-}ext \Rightarrow ('t,\ 'a,\ 'c)\ automaton\text{-}ext$
$\Rightarrow ('s \times 't,\ 'a,\ 'c)\ automaton\text{-}ext$ (**infix** $**$ *60*) **where**
$A ** B = (\!|$
$\delta = (\lambda\ (s,\ t)\ a\ .\ (\delta\ A\ s\ a,\ \delta\ B\ t\ a)),$
$Final = Final\ A \times Final\ B,$
$s_0 = (s_0\ A,\ \ s_0\ B),$
$\ldots = automaton.more\ A|\!)$

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-def*)

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

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

lemma [*simp*]: $\delta\ (A ** B)\ (s,\ t)\ a = (\delta\ A\ s\ a,\ \delta\ B\ t\ a)$
by (*simp add*: *product-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*: $\bigwedge A\ B\ .\ lang\ (A ** B)\ x = \ (lang\ A\ x \wedge lang\ B\ x)$
  apply (*induction x*)
  by (*auto simp add*: *lang-deltae*)

lemma *intersection*: $lang\ (A ** B) = \ (lang\ A \sqcap lang\ B)$
  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*: $-\ A = A(\!|Final := -Final\ A|\!)$
  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*]: $Final\ (-A) = -\ Final\ A$
  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*: $\bigwedge A\ .\ lang\ (-\ A)\ x = (\neg\ (lang\ A\ x))$
  by (*simp add*: *lang-deltae*)

lemma *complement*: $lang\ (-A) = (-\ (lang\ A))$
  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* = (|
  δ = (λ *u a* . (*case u of None* $\Rightarrow$ *None* | *Some s* $\Rightarrow$ *if s* $\in$ *Final A then None else Some* (δ *A s a*))),
  Final = *Some* ‘ (*Final A*) $\cup$ {*None*},
  s_0 = *Some* (s_0 *A*),
  ... = *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* $\Longrightarrow$ δ (*Prefix A*) (*Some s*) *a* = *None*
  by (*simp add*: *Prefix-def*)

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

lemma [*simp*]: *s* $\in$ *Final A* $\Longrightarrow$ *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*|)) = $\top$ $\Longrightarrow$ *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* $\Longrightarrow$ *y* $\leq$ *x* $\Longrightarrow$ *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_ψ A_φ $x = (\forall\ y\ .\ lang\ A_\psi\ (x\ @\ y) \longrightarrow (\exists\ z\ .(lang\ A_\varphi\ (x\ @\ z)) \wedge z \leq y))$

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

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

The weaker version of Urgency is introduced by:

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

lemma *Urgency-Urgency'-aux*: $(\forall\ y\ .\ lang\ A_\psi\ (x\ @\ y) \longrightarrow lang\ A_\varphi\ x) \Longrightarrow kfunc\ A_\psi\ A_\varphi\ x$
  by (*metis kfunc-def append-Nil2 less-eq-list.simps*(*1*))

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

lemma *Urgency-Urgency'*: *Urgency* A_ψ A_φ *Enf* $\Longrightarrow$ *Urgency'* A_ψ A_φ *Enf*
  by (*simp add*: *Urgency'-def Urgency-def Urgency-Urgency'-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-prediction*: $lang\ A_\psi = \top \Longrightarrow Urgency\ A_\psi\ A_\varphi\ Enf = (\forall x.\ lang\ A_\varphi\ x \longrightarrow Enf\ x = x)$
  apply (*auto simp add*: *Urgency-def kfunc-def*)
  apply (*metis append-Nil2 less-eq-list.simps*(*1*))
  by (*metis list.simps*(*4*) *neq-Nil-conv less-eq-list.simps*(*2*) *self-append-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-lang* A_ψ A_φ $x = (lang\ (A_\psi (\!| s_0 := (\delta e\ A_\psi\ x) |\!)) \leq lang\ ((Prefix\ A_\varphi)(\!|\ s_0 := Some\ (\delta e\ A_\varphi\ x)\ |\!)))$

lemma *kfunc-kfunc-lang*: *kfunc* A_ψ A_φ x = *kfunc-lang* A_ψ A_φ x
by (*simp add*: *kfunc-def kfunc-lang-def lang-deltaeb Predictive.Prefix-lang le-fun-def*)

lemma *kfunc-lang-empty*: *kfunc-lang* A_ψ A_φ x = (*lang* ((A_ψ ** − (*Prefix* A_φ))$(\!| s_0 := (\delta e\ A_\psi\ x,\ Some\ (\delta e\ A_\varphi\ x)) |\!)$) = $\bot$)
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_ψ A_φ x = (*lang* ((A_ψ ** − (*Prefix* A_φ))$(\!| s_0 := (\delta e\ A_\psi\ x,\ Some\ (\delta e\ A_\varphi\ x)) |\!)$) = $\bot$)
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-ext* ⇒ ('t, 'a, 'c) *automaton-ext* ⇒ 'a *list* ⇒ 'a *list* **where**
enforce A_ψ A_φ x =
(*if* x = [] *then*
[]
else
(*if kfunc* A_ψ A_φ x *then*
x
else
enforce A_ψ A_φ (*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_ψ A_φ $x \neq [] \Longrightarrow$ *kfunc* A_ψ A_φ (*enforce* A_ψ A_φ 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_ψ A_φ $y \Longrightarrow y \leq x \Longrightarrow y \leq$ (*enforce* A_ψ A_φ 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_φ $x \Longrightarrow$ *kfunc* A_ψ A_φ 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_ψ A_φ $x \leq x$
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_ψ A_φ *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_φ $x \Longrightarrow$ *enforce* A_ψ A_φ $x = x$
by (*simp add*: *lang-enf-kfunc*)

theorem *Urgency*: *kfunc* A_ψ A_φ $x \Longrightarrow$ *enforce* A_ψ A_φ $x = x$
by *simp*

theorem *Soundness*: *lang* A_ψ $x \Longrightarrow$ *enforce* A_ψ A_φ $x \neq [] \Longrightarrow$ *lang* A_φ (*enforce* A_ψ A_φ x)
proof −
assume A: *lang* A_ψ x
assume B: *enforce* A_ψ A_φ $x \neq []$
have *enforce* A_ψ A_φ $x \leq x$ **by** (*rule Transparency1*)
from *this* **obtain** z **where** D: x = *enforce* A_ψ A_φ x @ z **by** (*simp add*: *prefix-concat del*: *enforce.simps*, *safe*, *simp*)
from A **and** *this* **have** [*simp*]: *lang* A_ψ (*enforce* A_ψ A_φ x @ z) **by** *simp*
from B **have** *kfunc* A_ψ A_φ (*enforce* A_ψ A_φ x) **by** (*rule kfunc-enforce*)
from *this* **have** C: $\bigwedge y$. *lang* A_ψ (*enforce* A_ψ A_φ x @ y) $\Longrightarrow$ ($\exists t$. *lang* A_φ (*enforce* A_ψ A_φ x @ t) $\wedge$ $t \leq y$) **by** (*simp add*: *kfunc-def*)
have ($\exists t$. *lang* A_φ (*enforce* A_ψ A_φ x @ t) $\wedge$ ($t \leq z$)) **by** (*rule C*, *simp del*: *enforce.simps*)
then obtain *za* **where** F: *lang* A_φ (*enforce* A_ψ A_φ x @ *za*) **and** E: *za* $\leq z$

by *blast*
from *this* **have** *kfunc* A_ψ A_φ (*enforce* A_ψ A_φ *x* @ *za*) **by** (*simp add*: *lang-enf-kfunc del*: *enforce.simps*)
from *this* **have** *enforce* A_ψ A_φ *x* @ *za* $\leq$ *enforce* A_ψ A_φ *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* | *l1* | *l2*
datatype *Sig* = *a* | *b* | *c*
datatype *Sb* = *k0* | *k1* | *k2* | *k3*

fun
δa :: *Sa* $\Rightarrow$ *Sig* $\Rightarrow$ *Sa*
where
δa *l0 a* = *l0* |
δa *l0 b* = *l1* |
δa *l1 c* = *l0* |
δa - *a* = *l2* |
δa - *b* = *l2* |
δa - *c* = *l2*

definition *Fa* = {*l0*}

fun
δb :: *Sb* $\Rightarrow$ *Sig* $\Rightarrow$ *Sb*
where
δb *k0 a* = *k0* |
δb *k0 b* = *k1* |
δb *k1 a* = *k0* |
δb *k1 c* = *k2* |
δb *k2 a* = *k0* |
δb - *a* = *k3* |
δb - *b* = *k3* |
δb - *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] Authors omitted for blind review. Predictive Runtime Enforcement. Sept. 2015. Submitted.