Formalization of Predictive Runtime Enforcement

Viorel Preoteasa and Srivivas Pinisetty

December 11, 2015

Contents

theory *Predictive* **imports** *Main* **begin**

1 Languages as predicates on lists

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

Syntax for the lattice operations:

notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *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 $x \le y$. 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**
    $([] \le x) = True \mid$
    $((a \# x) \le y) = (case\ y\ of\ [] \Rightarrow False \mid b \# z \Rightarrow a = b \wedge (x \le z))$

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

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

  lemma *prefix-antisym*: $\bigwedge y\ .\ (x::'a\ list) \le y \Longrightarrow y \le x \Longrightarrow x = y$
    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*: $\bigwedge\ x\ .\ x \le y = (\exists\ z\ .\ y = x\ @\ z)$
    by (*induction* y, *simp-all*, *case-tac* x, *auto*)

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

  lemma *prefix-butlast*: $\bigwedge\ y\ .\ (x \le y) = (x = y \vee x \le (butlast\ y))$
    proof (*induction* x)
      case *Nil* **show** *?case* **by** *simp*
      case (*Cons x xs*)
        assume A: $\bigwedge\ y\ .\ (xs \le y) = (xs = y \vee xs \le (butlast\ y))$
        show *?case*
          apply *simp*
          apply (*case-tac* y, *simp-all*)
          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

2 Finite deterministic automata

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

primrec *lang*:: $('s, 'a, 'c)$ *automaton-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 δe:: $('s, 'a, 'b)$ *automaton-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*)

lemma *delta-but-last-aux*: $\bigwedge a\ s\ .\ \delta e\ (A(\!|s_0{:=}s|\!))\ (x\ @\ [a]) = \delta\ A\ (\delta e\ (A(\!|s_0{:=}s|\!))\ x)\ a$
apply (*induction* x)
by *simp-all*

lemma *delta-but-last*: $\delta e\ A\ (x\ @\ [a]) = \delta\ A\ (\delta e\ A\ x)\ a$
apply (*cut-tac* $A = A$ **and** $s = s_0\ A$ **and** $x = x$ **and** $a = a$ **in** *delta-but-last-aux*)

by *simp*

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-ext* $\Rightarrow$ $('t, 'a, 'c)$ *automaton-ext*
$\Rightarrow$ $('s \times 't, 'a, 'c)$ *automaton-ext* (**infix** $**$ *60*) **where**
$A ** B = (|$
$\delta = (\lambda\ (s,\ t)\ a\ .\ (\delta\ A\ s\ a,\ \delta\ B\ t\ a)),$
$Final = Final\ A \times Final\ B,$
$s_0 = (s_0\ A,\ \ s_0\ B),$
$\ldots = automaton.more\ A|)$

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*]: $\mathit{Final}\ (-A) = -\ \mathit{Final}\ A$
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*: $\bigwedge A\ .\ \mathit{lang}\ (-\ A)\ x = (\neg\ (\mathit{lang}\ A\ x))$
by (*simp add*: *lang-deltae*)

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

Next definition introduces an automaton *Extension A* based on automaton A. A list x is in the language of *Extension 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 $\varphi = \mathit{lang}\ A$.

definition *Extension* $A = (\!|$
$\delta = (\lambda\ s\ a\ .\ \mathit{if}\ s \in \mathit{Final}\ A\ \mathit{then}\ s\ \mathit{else}\ \delta\ A\ s\ a),$
$\mathit{Final} = \mathit{Final}\ A,$
$s_0 = s_0\ A,$
$\ldots = \mathit{more}\ A|\!)$

lemma [*simp*]: $s_0\ (\mathit{Extension}\ A) = s_0\ A$
by (*simp add*: *Extension-def*)

We introduce some properties of *Extension A* in the next six lemmas.

lemma *Extension-initial*[*simp*]: $\mathit{Extension}\ A(\!|s_0 := s|\!) = \mathit{Extension}\ (A(\!|s_0 := s|\!))$
by (*auto simp add*: *Extension-def fun-eq-iff*)

lemma *prefix-final*[*simp*]: $s \in \mathit{Final}\ A \Longrightarrow \mathit{lang}\ ((\mathit{Extension}\ A)(\!|s_0 := s|\!))\ x$
by (*induction x*, *simp-all add*: *Extension-def*)

lemma [*simp*]: $s_0\ A \in \mathit{Final}\ A \Longrightarrow \mathit{lang}\ (\mathit{Extension}\ A)\ x$

by (*drule prefix-final*, *simp*)

lemma [*simp*]: $s \in$ *Final* $A \Longrightarrow \delta$ (*Extension* A) $s\ a = s$
by (*simp add*: *Extension-def*)

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

lemma *lang* ((*Extension* A)$(\!|s_0 := s|\!)$) $= \top \Longrightarrow s \in$ *Final* A
apply (*simp add*: *fun-eq-iff Extension-def image-def*)
by (*drule-tac* $x = []$ **in** *spec*, *simp*)

The language of *Extension* A in terms of the language of A is given by the next lemma.

lemma *Extension-lang*: $\bigwedge$ (A::($'s$, $'a$, $'c$) *automaton-ext*) . *lang* (*Extension* A) x = ($\exists\ y$. *lang* $A\ y \wedge y \leq x$)
proof (*induction* x)
case (*Nil*) **show** *?case*
by (*simp add*: *Extension-def image-def*)
next
case (*Cons* $a\ x$)
assume A: $\bigwedge$ (A::($'s$, $'a$, $'c$) *automaton-ext*) . *lang* (*Extension* 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* (*Extension* A) x
by *blast*
show *?case*
apply *simp*
apply *safe*
apply (*case-tac* $s_0\ A \in$ *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* $s_0\ A \in$ *Final* A, *simp-all*)
by (*drule* B, *simp-all*)
qed

3 Constraints of the Predictive Enforcement

Next definition introduces $k_{\psi,\varphi}$ function from 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_ψ (x @ y) $\longrightarrow$ ($\exists\ z$.(*lang* A_φ (x @ z)) $\wedge\ z \leq y$))

The Urgency constraint is introduced in the next definition, and it has as hypothesis the $kfunc$ function. For simplicity we introduce first $kfunc$ and then Urgency. In the paper Urgency precedes the definition of $kfunc$.

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

definition $\mathit{Soundness}\ A_\psi\ A_\varphi\ \mathit{Enf} = (\forall\ x\ .\ \mathit{lang}\ A_\psi\ x \wedge \mathit{Enf}\ x \neq [] \longrightarrow \mathit{lang}\ A_\varphi\ (\mathit{Enf}\ x))$

definition $\mathit{Transparency1}\ \mathit{Enf} = (\forall\ x\ .\ \mathit{Enf}\ x \leq x)$

definition $\mathit{Transparency2}\ A_\varphi\ \mathit{Enf} = (\forall\ x\ .\ \mathit{lang}\ A_\varphi\ x \longrightarrow \mathit{Enf}\ x = x)$

definition $[\mathit{simp}]$: $\mathit{Monotonicity}\ \mathit{Enf} = \mathit{mono}\ \mathit{Enf}$

Next lemma shows that Transparency2 is a consequence of Urgency

lemma $\mathit{Urgency}\ A_\psi\ A_\varphi\ \mathit{Enf} \Longrightarrow \mathit{Transparency2}\ A_\varphi\ \mathit{Enf}$
by $(\mathit{metis}\ \mathit{Transparency2\text{-}def}\ \mathit{Urgency\text{-}def}\ \mathit{append\text{-}Nil2}\ \mathit{kfunc\text{-}def}\ \mathit{less\text{-}eq\text{-}list.simps}(1))$

4 Independence of Urgency, Transparency1, Monotonicity and Soundness

lemma $\mathit{Urgency\text{-}true}\ [\mathit{simp}]$: $\mathit{lang}\ A_\psi = \top \Longrightarrow \mathit{lang}\ A_\varphi = \bot \Longrightarrow \mathit{Urgency}\ A_\psi\ A_\varphi\ \mathit{Enf}$
by $(\mathit{simp}\ \mathit{add}\colon \mathit{Urgency\text{-}def}\ \mathit{kfunc\text{-}def})$

lemma $\mathit{Urgency\text{-}phi}[\mathit{simp}]$: $\mathit{lang}\ A_\psi = \top \Longrightarrow \mathit{lang}\ A_\varphi = B \Longrightarrow \mathit{Urgency}\ A_\psi\ A_\varphi\ \mathit{Enf} = (\forall\ x\ .\ B\ x \longrightarrow \mathit{Enf}\ x = x)$
apply $(\mathit{simp}\ \mathit{add}\colon \mathit{kfunc\text{-}def}\ \mathit{Urgency\text{-}def},\ \mathit{auto})$
apply $(\mathit{drule\text{-}tac}\ x = x$ **in** $\mathit{spec},\ \mathit{safe})$
apply $(\mathit{rule\text{-}tac}\ x = []$ **in** $\mathit{exI},\ \mathit{simp})$
apply $(\mathit{drule\text{-}tac}\ x = x$ **in** $\mathit{spec})$
by $(\mathit{drule\text{-}tac}\ x = []$ **in** $\mathit{spec},\ \mathit{simp})$

lemma $\mathit{Urgency\text{-}id}\ [\mathit{simp}]$: $\mathit{lang}\ A_\psi = \top \Longrightarrow \mathit{lang}\ A_\varphi = \top \Longrightarrow (\mathit{Urgency}\ A_\psi\ A_\varphi\ \mathit{Enf}) = (\mathit{Enf} = \mathit{id})$
by $(\mathit{simp}\ \mathit{add}\colon \mathit{Urgency\text{-}def}\ \mathit{kfunc\text{-}def}\ \mathit{fun\text{-}eq\text{-}iff},\ \mathit{auto})$

lemma $\mathit{Indep1}$: $\mathit{lang}\ A_\psi = \top \Longrightarrow \mathit{lang}\ A_\varphi = \bot \Longrightarrow (\forall\ x\ .\ \mathit{Enf}\ x = x) \Longrightarrow$
$\mathit{Urgency}\ A_\psi\ A_\varphi\ \mathit{Enf} \wedge \mathit{Monotonicity}\ \mathit{Enf} \wedge \mathit{Transparency1}\ \mathit{Enf} \wedge \neg\ \mathit{Soundness}\ A_\psi\ A_\varphi\ \mathit{Enf}$
by $(\mathit{simp}\ \mathit{add}\colon \mathit{Soundness\text{-}def}\ \mathit{mono\text{-}def}\ \mathit{Transparency1\text{-}def},\ \mathit{auto})$

lemma $\mathit{Indep1a}$: $\mathit{lang}\ A_\psi = \top \Longrightarrow \mathit{lang}\ A_\varphi = (\lambda\ x\ .\ x = [()]) \Longrightarrow (\forall\ x\ .\ \mathit{Enf}\ x = x) \Longrightarrow$
$\mathit{Urgency}\ A_\psi\ A_\varphi\ \mathit{Enf} \wedge \mathit{Monotonicity}\ \mathit{Enf} \wedge \mathit{Transparency1}\ \mathit{Enf} \wedge \neg\ \mathit{Soundness}\ A_\psi\ A_\varphi\ \mathit{Enf}$
by $(\mathit{simp}\ \mathit{add}\colon \mathit{Soundness\text{-}def}\ \mathit{mono\text{-}def}\ \mathit{Transparency1\text{-}def},\ \mathit{auto})$

lemma *Indep2*: *lang* A_ψ = ⊤ ⟹ *lang* A_φ = (λ *x* . *x* = [()]) ⟹ (∀ *x* . *Enf x* = [()]) ⟹
Urgency A_ψ A_φ *Enf* ∧ *Monotonicity Enf* ∧ ¬ *Transparency1 Enf* ∧ *Soundness* A_ψ A_φ *Enf*
apply (*simp add*: *Soundness-def mono-def Transparency1-def*)
by (*rule-tac x* = [] **in** *exI*, *simp*)

lemma *Indep3*: *lang* A_ψ = ⊤ ⟹ *lang* A_φ = (λ *x* . *x* = [()]) ⟹ (∀ *x* . *Enf x* = (*if x* = [()] *then* [()] *else* [])) ⟹
Urgency A_ψ A_φ *Enf* ∧ ¬ *Monotonicity Enf* ∧ *Transparency1 Enf* ∧ *Soundness* A_ψ A_φ *Enf*
apply (*simp add*: *Soundness-def mono-def Transparency1-def*)
by (*rule-tac x* = [(),()] **in** *exI*, *simp*)

lemma *Indep4*: *lang* A_ψ = ⊤ ⟹ *lang* A_φ = ⊤ ⟹ (∀ *x* . *Enf x* = []) ⟹
¬ *Urgency* A_ψ A_φ *Enf* ∧ *Monotonicity Enf* ∧ *Transparency1 Enf* ∧ *Soundness* A_ψ A_φ *Enf*
apply (*simp add*: *Soundness-def mono-def Transparency1-def*)
by (*rule-tac x* = [*Eps* ⊤] **in** *exI*, *simp*)

5 Alternative Urgency

A weaker version of Urgency is introduced by:

definition *Urgency′* A_ψ A_φ *Enf* = (∀ *x* . (∀ *y* . *lang* A_ψ (*x* @ *y*) ⟶ *lang* A_φ *x*) ⟶ *Enf x* = *x*)

lemma *Urgency-Urgency′-aux*: (∀ *y* . *lang* A_ψ (*x* @ *y*) ⟶ *lang* A_φ *x*) ⟹ *kfunc* A_ψ A_φ *x*
by (*metis kfunc-def append-Nil2 less-eq-list.simps*(*1*))

Urgency is stronger than Urgency’

lemma *Urgency-Urgency′*: *Urgency* A_ψ A_φ *Enf* ⟹ *Urgency′* A_ψ A_φ *Enf*
by (*metis Urgency′-def Urgency-def kfunc-def append-Nil2 less-eq-list.simps*(*1*))

6 Implementation of $kfunc$ as the inclusion of two regular languages

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_ψ⦇s_0 := (δ*e* A_ψ *x*)⦈) ≤ *lang* ((*Extension* A_φ)⦇ s_0 := δ*e* A_φ *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.Extension-lang le-fun-def*)

lemma *kfunc-lang-empty*: *kfunc-lang* A_ψ A_φ x = (*lang* ((A_ψ ** − (*Extension* A_φ))(|s_0 := (δe A_ψ x, δe A_φ x)|)) = ⊥)
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 ** − (*Extension* A_φ))(|s_0 := (δe A_ψ x, δe A_φ x)|)) = ⊥)
by (*unfold kfunc-kfunc-lang kfunc-lang-empty*, *simp*)

7 Enforcement Function

Next definition introduces the enforcement function. In this formalization we chose to define $enforce$ directly while in [1] we define it 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)))

When the property ψ is true for all sequences, then the Urgency property is simplified to the non-predictive case.

lemma *no-prediction*: *lang* A_ψ = ⊤ ⟹ *Urgency* A_ψ A_φ *Enf* = (∀ x. *lang* A_φ x ⟶ *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*)

When the property ψ is included in φ, then output of the enforcer is always equal to the input.

lemma *subset-enforce*: *lang* A_ψ ≤ *lang* A_φ ⟹ *enforce* A_ψ A_φ x = x
by (*metis enforce.simps kfunc-def order-refl predicate1D*)

When the property ψ is true for all sequences, then the kfun is simplified to:

lemma *no-prediction-kfunc*: *lang* A_ψ = ⊤ ⟹ *kfunc* A_ψ A_φ x = *lang* A_φ x
apply (*auto simp add*: *kfunc-def*)
using *less-eq-list.simps*(*1*) *prefix-antisym* **apply** *fastforce*
by (*metis append-Nil2 less-eq-list.simps*(*1*))

Next three lemmas are used in the proofs of soundness, transparency, and urgency.

lemma *kfunc-enforce*: $enforce\ A_\psi\ A_\varphi\ x \neq [] \Longrightarrow kfunc\ A_\psi\ A_\varphi\ (enforce\ A_\psi\ A_\varphi\ x)$
proof (*induction x rule*: *length-induct*)
fix $xs::'a\ list$
assume $\forall ys.\ length\ ys < length\ xs \longrightarrow enforce\ A_\psi\ A_\varphi\ ys \neq [] \longrightarrow kfunc\ A_\psi\ A_\varphi\ (enforce\ A_\psi\ A_\varphi\ ys)$
from *this* **have** A: $\bigwedge ys\ .\ length\ ys < length\ xs \Longrightarrow enforce\ A_\psi\ A_\varphi\ ys \neq [] \Longrightarrow kfunc\ A_\psi\ A_\varphi\ (enforce\ A_\psi\ A_\varphi\ ys)$
by *simp*
assume C: $enforce\ A_\psi\ A_\varphi\ xs \neq []$
from *this* **have** B: $xs \neq []$
by (*case-tac xs*, *simp-all*)
from B **and** C **have** D: $\neg\ kfunc\ A_\psi\ A_\varphi\ xs \Longrightarrow enforce\ A_\psi\ A_\varphi\ (butlast\ xs) \neq []$
apply (*subst enforce.simps*)
apply (*subst* (*asm*) *enforce.simps*)
apply (*subst* (*asm*) *enforce.simps*)
by (*simp del*: *enforce.simps*)
from B **and** D **show** $kfunc\ A_\psi\ A_\varphi\ (enforce\ A_\psi\ A_\varphi\ xs)$
apply (*subst enforce.simps*)
apply (*simp del*: *enforce.simps*, *safe*)
by (*cut-tac* $ys = butlast\ xs$ **in** A, *simp-all del*: *enforce.simps*)
qed

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

Finally we prove the enforcement function satisfies soundness, transparency, monotonicity, 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_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*)

lemma *Monotonicity-aux*: $\bigwedge x\ y\ .\ x \leq y \Longrightarrow n = \mathit{length}\ y \Longrightarrow \mathit{enforce}\ A_\psi\ A_\varphi\ x \leq \mathit{enforce}\ A_\psi\ A_\varphi\ y$
apply (*induction n*)
apply *simp*
apply (*subst enforce.simps*)
apply (*subst* (*2*) *enforce.simps*)
apply (*simp del*: *enforce.simps*, *safe*)
apply (*case-tac x*, *simp-all del*: *enforce.simps*)
apply (*metis kfunc-prefix-enforce prefix-butlast*)
apply (*metis Transparency1 enforce.simps prefix-trans*)
by (*metis Suc-eq-plus1 add.commute add-diff-cancel-left′ enforce.simps length-butlast prefix-butlast*)

theorem *Monotonicity*: *Monotonicity* (*enforce* A_ψ A_φ)
apply (*simp*)
apply (*unfold mono-def*)
apply *safe*
by (*rule Monotonicity-aux*, *simp-all*)

theorem *Urgency*: $\mathit{kfunc}\ A_\psi\ A_\varphi\ x \Longrightarrow \mathit{enforce}\ A_\psi\ A_\varphi\ x = x$
by *simp*

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

from *this* **have** *enforce* A_ψ A_φ *x* @ *za* $\le$ *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

8 Enforcement Algorithm

Because the enforcement algorithm has a non-terminating loop we cannot represent it as function in Isabelle. We introduce here the invariant of the algorithm, and we prove the initialization establishes the invariant, and that each step of the algorithm preserves the invariant. The invariant expresses the desired properties of the algorithm

x is the input received so far, y is the concatenation of all released words, and z (σ_c in the paper) is the pending word.

definition *Invariant* A_ψ A_φ *C p q x y z* = (
$C = A_\psi ** - (\mathit{Extension}\ A_\varphi)$
$\wedge\ p = \delta e\ A_\psi\ x$
$\wedge\ q = \ \delta e\ A_\varphi\ x \wedge x = y\ @\ z$
$\wedge$ *enforce* A_ψ A_φ $x = y$)

definition *Init* A_ψ A_φ = (*let* $(z, p, q, C) = ([], s_0\ A_\psi, s_0\ A_\varphi, A_\psi ** - (\mathit{Extension}\ A_\varphi))$ *in* (z, p, q, C))

definition *Step* A_ψ A_φ *C p q z a* = (*let* $(p',q') = (\delta\ A_\psi\ p\ a, \delta\ A_\varphi\ q\ a)$ *in* $(p',$ q', *if lang* $(C(\!|s_0 := (p',q')|\!)) = \bot$ *then* $(z\ @\ [a], [])$ *else* $([], z\ @\ [a])))$

The initialization establishes the invariant

lemma *Init*: (z, p, q, C) = *Init* A_ψ $A_\varphi \Longrightarrow$ *Invariant* A_ψ A_φ *C p q* [] [] *z*
by (*simp add*: *Init-def Invariant-def*)

The step preserves the invariant

lemma *Step*: (p', q', yr, zo) = *Step* A_ψ A_φ *C p q z a* $\Longrightarrow$ *Invariant* A_ψ A_φ *C p q x y z* $\Longrightarrow$ *Invariant* A_ψ A_φ *C* p' q' $(x\ @\ [a])$ $(y\ @\ yr)$ *zo*
apply (*simp add*: *Step-def*)
apply (*unfold Invariant-def*)
apply (*simp del*: *enforce.simps*)
apply (*cut-tac* $A_\psi 1 = A_\psi$ **and** $A_\varphi 1 = A_\varphi$ **and** $x1 = x\ @\ [a]$ **in** *kfunc-lang-empty* [*THEN sym*])
apply (*simp del*: *enforce.simps add*: *delta-but-last*)
apply (*unfold append-assoc* [*THEN sym*])
apply (*simp del*: *enforce.simps append-assoc add*: *delta-but-last*)
apply (*unfold kfunc-kfunc-lang* [*THEN sym*])
apply (*case-tac kfunc* A_ψ A_φ $((y\ @\ z)\ @\ [a])$)
apply (*simp-all del*: *enforce.simps*)

apply (*rule Urgency*, *simp*)
by (*metis* (*no-types*, *lifting*) *append-assoc enforce.simps snoc-eq-iff-butlast*)

9 Example

datatype *Sa* = *l0* | *l1* | *l2*
datatype *Sig* = *a* | *b* | *c*
datatype *Sb* = *k0* | *k1* | *k2* | *k3*

fun
δa :: *Sa* ⇒ *Sig* ⇒ *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* ⇒ *Sig* ⇒ *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 Extension-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] S. Pinisetty, V. Preoteasa, S. Tripakis, T. Jéron, Y. Falcone, and H. Marchand. Predictive Runtime Enforcement. Sept. 2015. Submitted.