thy

By artur

February 8, 2017

Contents

theory *Prelude*
imports *Main*
begin

You can place here what you want. However, in practice it is recommended to restrict additions here only to ingredients of the Haskell Prelude; further Haskell library modules should be obtained in source and just imported, probably with prior modifications.

0.1 Equality

class $eq =$
  fixes $eq :: 'a \Rightarrow 'a \Rightarrow bool$
  fixes $not\text{-}eq :: 'a \Rightarrow 'a \Rightarrow bool$
  assumes $not\text{-}eq$ $[simp]$: $not\text{-}eq\ x\ y \longleftrightarrow \neg\ eq\ x\ y$

instantiation $bool :: eq$
begin

definition
  $eq\ p\ q \longleftrightarrow (p \longleftrightarrow q)$

definition
  $not\text{-}eq\ p\ q \longleftrightarrow \neg\ (p \longleftrightarrow q)$

instance proof

qed (*simp-all add*: *eq-bool-def not-eq-bool-def*)

end

instantiation *unit* :: *eq*
begin

definition
$eq\ (u{::}unit)\ v \longleftrightarrow True$

definition
$not\text{-}eq\ (u{::}unit)\ v \longleftrightarrow False$

instance proof
qed (*simp-all add*: *eq-unit-def not-eq-unit-def*)

end

instantiation *prod* :: (*eq*, *eq*) *eq*
begin

definition
$eq\ x\ y \longleftrightarrow (x :: \text{-} * \text{-}) = y$

definition
$not\text{-}eq\ x\ y \longleftrightarrow (x :: \text{-} * \text{-}) \neq y$

instance proof
qed (*simp-all add*: *eq-prod-def not-eq-prod-def*)

end

instantiation *list* :: (*eq*) *eq*
begin

definition
$eq\ x\ y \longleftrightarrow (x :: \text{-}\ list) = y$

definition
$not\text{-}eq\ x\ y \longleftrightarrow (x :: \text{-}\ list) \neq y$

instance proof
qed (*simp-all add*: *eq-list-def not-eq-list-def*)

end

instantiation *option* :: (*eq*) *eq*
begin

definition
$\mathit{eq}\ x\ y \longleftrightarrow (x :: \text{-}\ \mathit{option}) = y$

definition
$\mathit{not\text{-}eq}\ x\ y \longleftrightarrow (x :: \text{-}\ \mathit{option}) \neq y$

instance proof
qed (*simp-all add*: *eq-option-def not-eq-option-def*)

end

instantiation *int* :: *eq*
begin

definition
$\mathit{eq}\ x\ y \longleftrightarrow x = (y{::}\mathit{int})$

definition
$\mathit{not\text{-}eq}\ x\ y \longleftrightarrow x \neq (y{::}\mathit{int})$

instance proof
qed (*simp-all add*: *eq-int-def not-eq-int-def*)

end

0.2 Fundamental prelude ingredients

axiomatization $\mathit{error} :: \mathit{string} \Rightarrow {'a}$

abbreviation (*input*) $\mathit{rapp} :: ({'a} \Rightarrow {'b}) \Rightarrow {'a} \Rightarrow {'b}$ (**infixr** \$ *60*) **where**
$f\ \$\ x \equiv f\ x$

abbreviation (*input*) $\mathit{const} :: {'a} \Rightarrow {'b} \Rightarrow {'b}$ **where**
$\mathit{const}\ x\ y \equiv y$

definition $\mathit{curry} :: ({'a} \times {'b} \Rightarrow {'c}) \Rightarrow {'a} \Rightarrow {'b} \Rightarrow {'c}$ **where**
$\mathit{curry}\ f\ x\ y = f\ (x,\ y)$

0.3 Options

definition $\mathit{the\text{-}default} :: {'a} \Rightarrow {'a}\ \mathit{option} \Rightarrow {'a}$ **where**
$\mathit{the\text{-}default}\ x\ y = (\mathit{case}\ y\ \mathit{of}\ \mathit{Some}\ z \Rightarrow z \mid \mathit{None} \Rightarrow x)$

abbreviation (*input*) $\mathit{maybe} :: {'b} \Rightarrow ({'a} \Rightarrow {'b}) \Rightarrow {'a}\ \mathit{option} \Rightarrow {'b}$ **where**
$\mathit{maybe}\ x\ f\ y \equiv \mathit{the\text{-}default}\ x\ (\mathit{map\text{-}option}\ f\ y)$

0.4 Lists

abbreviation (*input*) $\mathit{null} :: {'a}\ \mathit{list} \Rightarrow \mathit{bool}$ **where**
$\mathit{null}\ \mathit{xs} \equiv \mathit{xs} = []$

definition *nth* :: *'a list* ⇒ *int* ⇒ *'a* **where**
  nth xs k = (*if k* < *0 then error ''negative index'' else List.nth xs* (*nat k*))

definition *length* :: *'a list* ⇒ *int* **where**
  length xs = *int* (*List.length xs*)

definition *replicate* :: *int* ⇒ *'a* ⇒ *'a list* **where**
  replicate k = *List.replicate* (*nat k*)

primrec *separate* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
    separate x [] = []
  | *separate x* (*y* # *ys*) = (*if ys* = [] *then* [*y*] *else y* # *x* # *separate x ys*)

0.5 Counterparts for fundamental Haskell classes

class *ord* = *eq* + *linorder*

instance *int* :: *ord* **..**

class *print* =
  fixes *print* :: *'a* ⇒ *string*

instantiation *list* :: (*print*) *print*
begin

definition
  print xs = ''['' @ *concat* (*separate* '', '' (*map print xs*)) @ '']''

instance ..

end

class *num* = *comm-ring-1* + *abs* + *sgn* + *eq* + *print*

instance *int* :: *num* **..**

end
theory *AST*
imports *Prelude*
begin

type-synonym *GivenValue* = *string*

type-synonym *ZInt* = *int*

type-synonym *ZDecor* = *string*

```
type-synonym ZVar = string * (ZDecor list)

type-synonym GivenSet = ZVar

type-synonym ZName = string

datatype ZGenFilt = Choose ZVar ZExpr
                  | Check ZPred
                  | Evaluate ZVar ZExpr ZExpr
and      ZExpr = ZVar ZVar
               | ZInt ZInt
               | ZTuple ZExpr list
               | ZBinding (ZVar * ZExpr) list
               | ZSetDisplay ZExpr list
               | ZSeqDisplay ZExpr list
               | ZCross ZExpr list
               | ZSetComp ZGenFilt list ZExpr option
               | ZCall ZExpr ZExpr
and      ZPred = ZFalse ZPred list
               | ZTrue ZPred list
               | ZAnd ZPred ZPred
               | ZPSchema ZSExpr
and      ZSExpr = ZSchema ZGenFilt list

primrec reason :: ZPred ⇒ ZPred list
where
  reason (ZTrue x) = x
| reason (ZFalse x) = x


primrec update-reason :: ZPred list ⇒ ZPred ⇒ ZPred
where
  update-reason x (ZTrue -) = (ZTrue x)
| update-reason x (ZFalse -) = (ZFalse x)


type-synonym ZFSet = ZExpr list

datatype ZSName = ZSPlain string
                | ZSDelta string
                | ZSXi string

datatype CSExp = CSExpr ZName
               | CSEmpty
               | CChanSet ZName list
               | ChanSetUnion CSExp CSExp
               | ChanSetInter CSExp CSExp
               | ChanSetDiff CSExp CSExp
```

```
datatype NSExp = NSExpEmpty
               | NSExprMult ZName list
               | NSExprSngl ZName
               | NSExprParam ZName ZExpr list
               | NSUnion NSExp NSExp
               | NSIntersect NSExp NSExp
               | NSHide NSExp NSExp
               | NSBigUnion ZExpr

datatype CParameter = ChanInp ZName
                    | ChanInpPred ZName ZPred
                    | ChanOutExp ZExpr
                    | ChanDotExp ZExpr

datatype Comm = ChanComm ZName CParameter list
              | ChanGenComm ZName ZExpr list CParameter list

datatype CAction = CSPSkip
                 | CSPStop
                 | CSPChaos
                 | CSPCommAction Comm CAction
                 | CSPSeq CAction CAction
                 | CSPExtChoice CAction CAction

datatype ParAction = CircusAction CAction
                   | ParamActionDecl ZGenFilt list ParAction

datatype ZPara = ZSchemaDef ZSName ZSExpr
               | Process ProcDecl
and      ProcDecl = CProcess ZName ProcessDef
                  | CParamProcess ZName ZName list ProcessDef
                  | CGenProcess ZName ZName list ProcessDef
and      ProcessDef = ProcDefSpot ZGenFilt list ProcessDef
                    | ProcDef CProc
and      CProc = ProcMain ZPara PPar list CAction
               | ProcStalessMain PPar list CAction
and      PPar = ProcZPara ZPara
              | CParAction ZName ParAction
              | CNameSet ZName NSExp

type-synonym ZSpec = ZPara list

type-synonym CProgram = ZPara list

end
theory OmegaDefs
imports AST Prelude
begin
```

fun *join-name*
where
join-name n v = (*n* @ (''-'' @ *v*))

fun *free-var-ZGenFilt*
where
free-var-ZGenFilt (*Choose v e*) = [*v*]
| *free-var-ZGenFilt* (*Check p*) = *Nil*
| *free-var-ZGenFilt* (*Evaluate v e1 e2*) = *Nil*

fun *free-var-ZPred* :: *ZPred* ⇒ *ZVar list*
where
free-var-ZPred (*ZFalse p*) = *Nil*
| *free-var-ZPred* (*ZTrue p*) = *Nil*
| *free-var-ZPred* (*ZAnd a b*) = (*free-var-ZPred a* @ *free-var-ZPred b*)
| *free-var-ZPred x* = *Nil*

fun *fvs*
where
fvs f Nil = *Nil*
| *fvs f* (*e* # *es*) = (*f e* @ (*fvs f es*))

function (*sequential*) *free-var-ZExpr* :: *ZExpr* ⇒ *ZVar list*
where
free-var-ZExpr (*ZVar v*) = [*v*]
| *free-var-ZExpr* (*ZInt c*) = *Nil*
| *free-var-ZExpr* (*ZSetDisplay exls*) = *fvs free-var-ZExpr exls*
| *free-var-ZExpr* (*ZSeqDisplay exls*) = *fvs free-var-ZExpr exls*
| *free-var-ZExpr* (*ZCall ex ex2*) = *free-var-ZExpr ex2*
| *free-var-ZExpr* - = *Nil*
by *pat-completeness auto*

fun *make-get-com* :: *ZName list* ⇒ *CAction* ⇒ *CAction*
where
make-get-com [*x*] *c* = (*CSPCommAction* (*ChanComm* ''*mget*'' [*ChanDotExp* (*ZVar* (*x*, *Nil*)), *ChanInp* (''*v-*'' @ *x*)]) *c*)
| *make-get-com* (*x* # *xs*) *c* = (*CSPCommAction* (*ChanComm* ''*mget*'' [*ChanDotExp* (*ZVar* (*x*, *Nil*)), *ChanInp* (''*v-*'' @ *x*)]) (*make-get-com xs c*))
| *make-get-com x c* = *c*

fun *make-set-com* :: (*CAction* ⇒ *CAction*) ⇒ *ZVar list* ⇒ *ZExpr list* ⇒ *CAction* ⇒ *CAction*
where
make-set-com f [(*x*, -)] [*y*] *c* = (*CSPCommAction* (*ChanComm* ''*mset*'' [*ChanDotExp*

(*ZVar* (*x*, *Nil*)), *ChanOutExp y*]) (*f c*))
| *make-set-com f* ((*x*, -) # *xs*) (*y* # *ys*) *c* = (*CSPCommAction* (*ChanComm* ''*mset*'' [*ChanDotExp* (*ZVar* (*x*, *Nil*)), *ChanOutExp y*]) (*make-set-com f xs ys c*))
| *make-set-com f* - - *c* = (*f c*)

fun *getWrtV*
where
getWrtV xs = *Nil*

fun *rename-ZPred*
where
rename-ZPred (*ZFalse a*) = (*ZFalse a*)
| *rename-ZPred* (*ZTrue a*) = (*ZTrue a*)
| *rename-ZPred* (*ZAnd p1 p2*) = (*ZAnd* (*rename-ZPred p1*) (*rename-ZPred p2*))
| *rename-ZPred* (*ZPSchema sp*) = (*ZPSchema sp*)

fun *inListVar*
where
```
  inListVar x Nil = False
| inListVar x [va] = (case x = va of
                        True ⇒ True
                      | - ⇒ False)
| inListVar x (va # vst) = (case x = va of
                              True ⇒ True
                            | - ⇒ inListVar x vst)
```

fun *delete-from-list*
where
```
  delete-from-list x Nil = Nil
| delete-from-list x [v] = (case x = v of
                              True ⇒ Nil
                            | False ⇒ [v])
| delete-from-list x (v # va) = (case x = v of
                                   True ⇒ delete-from-list x va
                                 | False ⇒ (v # (delete-from-list x va)))
```

fun *setminus*
where
setminus Nil - = *Nil*
| *setminus* (*v* # *va*) *Nil* = (*v* # *va*)
| *setminus* (*v* # *va*) (*b* # *vb*) = ((*delete-from-list b* (*v* # *va*)) @ (*setminus* (*v* # *va*) *vb*))

fun *member*

where
  member x Nil = False
| *member x (b # y) = (if x = b then True else member x y)*

fun *intersect*
where
  intersect Nil y = Nil
| *intersect (a # x) y = (if member a y then a # (intersect x y)*
else intersect x y)

fun *union*
where
  union Nil y = y
| *union (a # x) y = (if (member a y) then (union x y)*
else a # (union x y))

fun *elem-by* :: *'a ⇒ 'a list ⇒ bool*
where
  elem-by - Nil = False
| *elem-by y (x # xs) = (y = x | elem-by y xs)*

fun *isPrefixOf*
where
  isPrefixOf Nil - = True
| *isPrefixOf - Nil = False*
| *isPrefixOf (x # xs) (y # ys) = (x = y & isPrefixOf xs ys)*

fun *get-ZVar-st*
where
  get-ZVar-st (a, x) = (case (isPrefixOf ''st-var-'' a) of
True ⇒ [a]
| False ⇒ Nil)

fun *free-var-CAction* :: *CAction ⇒ ZVar list*
where
  free-var-CAction CSPSkip = Nil
| *free-var-CAction CSPStop = Nil*
| *free-var-CAction CSPChaos = Nil*
| *free-var-CAction (CSPSeq ca cb) = ((free-var-CAction ca) @ (free-var-CAction cb))*
| *free-var-CAction (CSPExtChoice ca cb) = ((free-var-CAction ca) @ (free-var-CAction cb))*
| *free-var-CAction (CSPCommAction v va) = Nil*

fun *is-ZVar-st*
where
  is-ZVar-st a = isPrefixOf ''st-var-'' a

fun *rename-ZExpr*
where
  rename-ZExpr (ZVar (va, x)) = (case (is-ZVar-st va) of
  True ⇒ (ZVar (''v-'' @ va, x))
  | False ⇒ (ZVar (va, x)))
| rename-ZExpr (ZInt zi) = (ZInt zi)
| rename-ZExpr (ZCall xpr1 xpr2) = (ZCall (rename-ZExpr xpr1) (rename-ZExpr xpr2))
| rename-ZExpr x = x

fun *bindingsVar*
where
  bindingsVar Nil = Nil
| bindingsVar [((va, x), b)] = (case (is-ZVar-st va) of
  True ⇒ [((''v-'' @ va, x), (rename-ZExpr b))]
  | False ⇒ [((va, x), (rename-ZExpr b))])
| bindingsVar (((va, x), b) # xs) = (case (is-ZVar-st va) of
  True ⇒ [((''v-'' @ va, x), (rename-ZExpr b))] @ (bindingsVar xs)
  | False ⇒ [((va, x), (rename-ZExpr b))] @ (bindingsVar xs))

fun *rename-vars-CAction*
where
  rename-vars-CAction CSPSkip = CSPSkip
| rename-vars-CAction CSPStop = CSPStop
| rename-vars-CAction CSPChaos = CSPChaos
| rename-vars-CAction (CSPSeq a1 a2) = (CSPSeq (rename-vars-CAction a1) (rename-vars-CAction a2))
| rename-vars-CAction (CSPExtChoice a1 a2) = (CSPExtChoice (rename-vars-CAction a1) (rename-vars-CAction a2))
| rename-vars-CAction x = x

fun *remdups*
where
  remdups Nil = Nil
| remdups (x # xs) = (if (member x xs) then remdups xs
  else x # remdups xs)

fun *getFV*
where
getFV xs = *Nil*

fun *subset*
where
subset xs ys = *list-all* (% *arg0* . *member arg0 ys*) *xs*

end
theory *MappingFunStatelessCircus*
imports *AST OmegaDefs Prelude*
begin

fun *omega-CAction* :: *CAction* ⇒ *CAction* **and**
omega-prime-CAction :: *CAction* ⇒ *CAction*
where
omega-CAction CSPSkip = *CSPSkip*
| *omega-CAction CSPStop* = *CSPStop*
| *omega-CAction CSPChaos* = *CSPChaos*
| *omega-CAction* (*CSPSeq ca cb*) = (*CSPSeq* (*omega-CAction ca*) (*omega-CAction cb*))
| *omega-CAction* (*CSPExtChoice ca cb*) = (*let lsx* = (*map fst* (*remdups* (*free-var-CAction* (*CSPExtChoice ca cb*))))
in make-get-com lsx (*rename-vars-CAction* (*CSPExtChoice* (*omega-prime-CAction ca*) (*omega-prime-CAction cb*))))
| *omega-CAction x* = *x*
| *omega-prime-CAction CSPSkip* = *CSPSkip*
| *omega-prime-CAction CSPStop* = *CSPStop*
| *omega-prime-CAction CSPChaos* = *CSPChaos*
| *omega-prime-CAction* (*CSPSeq ca cb*) = (*CSPSeq* (*omega-prime-CAction ca*) (*omega-prime-CAction cb*))
| *omega-prime-CAction x* = *omega-CAction x*

end