

Chapter 1

Library LNMItpred

LNMItpred.v Version 2.7 January 2010 does not need impredicative Set, runs under V8.2, tested with version 8.2pl1

Copyright Ralph Matthes, I.R.I.T., C.N.R.S. & University of Toulouse

provides basic definitions and the predicative specification of LNMItpred and infers general theorems from this specification.

this is code that no longer conforms to the description in the article "An induction principle for nested datatypes in intensional type theory" by the author, that appeared in the Journal of Functional Programming, since it now uses type classes instead of the record **EFct** and the type **pEFct**, as well as for **mon** and **NAT**

forms part of the code that comes with a submission to the journal Science of Computer Programming

Require Import Utf8.

the universe of all monotypes:

Definition k0 := Set.

the type of all type transformations:

Definition k1 := k0 → k0.

the type of all rank-2 type transformations:

Definition k2 := k1 → k1.

polymorphic identity:

Definition id : ∀ (A: Set), A → A := fun A x => x.

Implicit Arguments id [[A]].

composition:

Definition comp (A B C: Type)(g: B → C)(f: A → B) : A → C := fun x => g (f x).

comp is displayed as the infix \circ .

standard notion of less than on type transformations, in `Type` and not in `Set` that would require impredicative `Set`

Definition `sub_k1 (X Y: k1) : Type := $\forall A: \text{Set}, X A \rightarrow Y A$.`

`sub_k1` is displayed as the infix \subseteq .

monotonicity:

Class `mon (X: k1) : Type :=`

`map: $\forall (A B: \text{Set}), (A \rightarrow B) \rightarrow X A \rightarrow X B$.`

extensionality:

Definition `ext (X: k1){mX: mon X}: Prop :=`

`$\forall (A B: \text{Set})(f g: A \rightarrow B), (\forall a, f a = g a) \rightarrow \forall r, \text{map } f r = \text{map } g r$.`

Require Import Setoid.

Require Import Morphisms.

Definition `ext' (X: k1){mX: mon X}: Prop :=`

`$\forall (A B: \text{Set}), \text{Morphism } ((@eq A ==> @eq B) ==> @eq (X A) ==> @eq (X B))$
(map(A:= A)(B:= B)).`

Lemma `extEquiv (X: k1){mX: mon X}: ext \leftrightarrow ext'.`

first functor law:

Definition `fct1 (X: k1){mX: mon X}: Prop :=`

`$\forall (A: \text{Set})(x: X A), \text{map id } x = x$.`

second functor law:

Definition `fct2 (X: k1){mX: mon X}: Prop :=`

`$\forall (A B C: \text{Set})(f: A \rightarrow B)(g: B \rightarrow C)(x: X A), \text{map } (g \circ f) x = \text{map } g (\text{map } f x)$.`

pack up the good properties of the approximation into the notion of an extensional functor:

Class `EFct (X: k1) := {m :> mon X; e: ext; f1: fct1; f2: fct2}`.

preservation of extensional functors:

Class `pEFct (F: k2) := pres:> $\forall (X: k1), \text{EFct } X \rightarrow \text{EFct } (F X)$.`

`>` was suggested to me by Matthieu Sozeau on November 17, 2008

we show some closure properties of `pEFct`, depending on such properties for `EFct`

Instance `moncomp '{X: k1, mX: mon X, Y: k1, mY: mon Y}: mon (fun A \Rightarrow X(Y A)).`

closure under composition:

Instance `compEFct '{X: k1, EFct X, Y: k1, EFct Y} : EFct (fun A \Rightarrow X(Y A)):= {m:= moncomp(mX:= map)(mY:= map)}.`

Instance `comppEFct '{F: k2, pEFct F, G: k2, pEFct G}: pEFct (fun X A \Rightarrow F X (G X A)).`

This may now be used to prove that truly nested examples are instances of the theorems to come.

closure under sums:

```
Instance sumEFct '{X: k1, EFct X, Y: k1, EFct Y}: EFct (fun A => X A + Y A)%type:=
  {m:= (fun A B f x => match x with
    | inl y => inl _ (map f y)
    | inr y => inr _ (map f y)
  end): mon (fun A : Set => (X A + Y A)%type)}.
```

```
Instance sumpEFct '{F: k2, pEFct F, G: k2, pEFct G}: pEFct (fun X A => F X A + G
X A)%type.
```

closure under products:

```
Instance prodEFct '{X: k1, EFct X, Y: k1, EFct Y}: EFct (fun A => X A × Y A)%type:=
  {m:= (fun A B f x => match x with
    (x1, x2) => (map f x1, map f x2) end): mon(fun A => X A × Y A)%type}.
```

```
Instance prodpEFct '{F: k2, pEFct F, G: k2, pEFct G}: pEFct (fun X A => F X A × G
X A)%type.
```

the identity in k2 preserves extensional functors:

```
Instance idpEFct: pEFct (fun X => X).
```

a variant for the eta-expanded identity:

```
Instance idpEFct_eta: pEFct (fun X A => X A).
```

Is there any possibility to avoid the destruct command?

the identity in k1 "is" an extensional functor:

```
Instance idEFct: EFct (fun A => A) := {m:= fun A B (f: A → B)(x: A) => f x}.
```

constants in k2:

```
Instance constpEFct '{X: k1, EFct X}: pEFct (fun _ => X).
```

constants in k1:

```
Instance constEFct(C: Set): EFct (fun _ => C):= {m:= fun A B (f: A → B)(x: C) =>
x}.
```

the option type:

```
Instance optionEFct: EFct (fun A: Set => option A):= {m:= option_map}.
```

Defined.

natural transformations from (X, mX) to (Y, mY) :

```
Class NAT(X Y: k1)(j: X ⊆ Y){mX: mon X, mY: mon Y} : Prop :=
  naturality: ∀ (A B: Set)(f: A → B)(t: X A), j B (map f t) = map f (j A t).
```

a notion that plays a role in the uniqueness theorem for Mlt:

```
Definition polyExtsub (X1 X2 Y1 Y2: k1)(t: X1 ⊆ X2 → Y1 ⊆ Y2) : Prop :=
```

$\forall (f\ g: X_1 \subseteq X_2)(A: \text{Set})(y: Y_1\ A),$
 $(\forall (A: \text{Set})(x: X_1\ A), f\ A\ x = g\ A\ x) \rightarrow t\ f\ A\ y = t\ g\ A\ y.$

Definition `polyExtsub'` $(X_1\ X_2\ Y_1\ Y_2: \mathbf{k1}): (X_1 \subseteq X_2 \rightarrow Y_1 \subseteq Y_2) \rightarrow \mathbf{Prop} :=$
Morphism $(\text{forall_relation } (\text{fun } A: \text{Set} \Rightarrow @eq\ (X_1\ A) ==> @eq\ (X_2\ A)) ==>$
 $\text{forall_relation } (\text{fun } A: \text{Set} \Rightarrow @eq\ (Y_1\ A) ==> @eq\ (Y_2\ A))).$

Lemma `polyExtsubEquiv` $(X_1\ X_2\ Y_1\ Y_2: \mathbf{k1})(t: X_1 \subseteq X_2 \rightarrow Y_1 \subseteq Y_2):$
`polyExtsub` $t \leftrightarrow \text{polyExtsub}'\ t.$

a notion of monotonicity that is needed for "canonization": relativized basic monotonicity of rank 2

Definition `mon2br` $(F: \mathbf{k2}) := \forall (X\ Y: \mathbf{k1}), \mathbf{mon}\ Y \rightarrow X \subseteq Y \rightarrow F\ X \subseteq F\ Y.$

the predicative specification of *LNMIT*:

Module Type `LNMIT_TYPE`.

Parameter $F: \mathbf{k2}$.

Instance `FpEFct`: **pEFct** F .

Parameter $\mu_0: \mathbf{k1}$.

Definition $\mu: \mathbf{k1} := \text{fun } A \Rightarrow \mu_0\ A.$

the introduction of μ_0 is a little twist that is mentioned in the paper, but only in a footnote and not relevant there

Instance `mapmu2`: **mon** μ . Definition `MltType`: `Type` :=
 $\forall G : \mathbf{k1}, (\forall X : \mathbf{k1}, X \subseteq G \rightarrow F\ X \subseteq G) \rightarrow \mu \subseteq G.$

Parameter `Mlt0` : `MltType`. we need just its eta-expansion `Mlt`

Definition `Mlt` : `MltType` := `fun G s A t => Mlt0 s t.`

Definition `lnType` : `Type` :=

$\forall (X: \mathbf{k1})(ef: \mathbf{EFct}\ X)(j: X \subseteq \mu), \mathbf{NAT}\ j \rightarrow F\ X \subseteq \mu.$

Parameter `ln` : `lnType`.

Axiom `mapmu2Red` : $\forall (A: \text{Set})(X: \mathbf{k1})(ef: \mathbf{EFct}\ X)(j: X \subseteq \mu)$
 $(n: \mathbf{NAT}\ j)(t: F\ X\ A)(B: \text{Set})(f: A \rightarrow B),$
 $\text{mapmu2}\ f\ (\text{ln}\ ef\ n\ t) = \text{ln}\ ef\ n\ (\text{m}\ f\ t).$

Axiom `MltRed` : $\forall (G: \mathbf{k1})$

$(s: \forall X: \mathbf{k1}, X \subseteq G \rightarrow F\ X \subseteq G)(X: \mathbf{k1})(ef: \mathbf{EFct}\ X)(j: X \subseteq \mu)$
 $(n: \mathbf{NAT}\ j)(A: \text{Set})(t: F\ X\ A),$
 $\text{Mlt}\ s\ (\text{ln}\ ef\ n\ t) = s\ X\ (\text{fun } A \Rightarrow (\text{Mlt}\ s\ (A := A)) \circ (j\ A))\ A\ t.$

Definition `mu2lndType` : `Prop` :=

$\forall (P: (\forall A: \text{Set}, \mu\ A \rightarrow \mathbf{Prop})),$
 $(\forall (X: \mathbf{k1})(ef: \mathbf{EFct}\ X)(j: X \subseteq \mu)(n: \mathbf{NAT}\ j),$
 $(\forall (A: \text{Set}) (x: X\ A), P\ A\ (j\ A\ x)) \rightarrow$
 $\forall (A: \text{Set})(t: F\ X\ A), P\ A\ (\text{ln}\ ef\ n\ t)) \rightarrow$
 $\forall (A: \text{Set}) (r: \mu\ A), P\ A\ r.$

Axiom `mu2lnd` : `mu2lndType`.

End LNMIT_TYPE.

prove theorems about the components of LNMIT_TYPE

Module LNMITDEF(M :LNMIT_Type).

Export M .

we complete `mapmu2` to an extensional functor; the order of the following three lemmas does not play a role; they only make shallow use of the induction principle in that they do not use the induction hypothesis

Lemma `mapmu2Ext`: $\text{ext}(mX := \text{mapmu2})$.

Lemma `mapmu2ld`: $\text{fct1}(mX := \text{mapmu2})$.

Lemma `mapmu2Comp`: $\text{fct2}(mX := \text{mapmu2})$.

Instance `mapmu2EFct`: $\mathbf{EFct} \mu := \{m := \text{mapmu2}\}$.

Instance `mapmu2NAT`(X : $\mathbf{k1}$)(ef : $\mathbf{EFct} X$)(j : $X \subseteq \mu$)(n : $\mathbf{NAT} j$): $\mathbf{NAT} (\text{In } ef \ n)$.

the canonical constructor for μ uses μ as its own approximation:

Definition `InCan` : $F \mu \subseteq \mu :=$

$\text{fun } A \ t \Rightarrow \text{In } \text{mapmu2EFct} \ (j := \text{fun } _ \ x \Rightarrow x)(\text{fun } _ \ _ \ _ \Rightarrow \text{refl_equal } _)$ t .

the behaviour for canonical elements:

Lemma `MltRedCan` : $\forall (G : \mathbf{k1})(s : \forall X : \mathbf{k1}, X \subseteq G \rightarrow F X \subseteq G)$
 $(A : \mathbf{Set})(t : F \mu A)$, $\text{Mlt } s \ (\text{InCan } t) = s \ \mu \ (\text{Mlt } s) \ A \ t$.

Lemma `mapmu2RedCan` : $\forall (A : \mathbf{Set})(B : \mathbf{Set})(f : A \rightarrow B)(t : F \mu A)$,
 $\text{mapmu2 } f \ (\text{InCan } t) = \text{InCan}(m \ f \ t)$.

Instance `mapmu2NATCan`: $\mathbf{NAT} \ \text{InCan}$.

`MltRed` already characterizes `Mlt s` under an extensionality assumption for s :

Theorem `MltUni`: $\forall (G : \mathbf{k1})(s : \forall X : \mathbf{k1}, X \subseteq G \rightarrow F X \subseteq G)(h : \mu \subseteq G)$,
 $(\forall (X : \mathbf{k1}), \text{polyExtsub}(s \ X)) \rightarrow$
 $(\forall (X : \mathbf{k1})(ef : \mathbf{EFct} X)(j : X \subseteq \mu)(n : \mathbf{NAT} j)(A : \mathbf{Set})(t : F X A)$,
 $h \ A \ (\text{In } ef \ n \ t) = s \ X \ (\text{fun } A \Rightarrow (h \ A) \circ (j \ A)) \ A \ t) \rightarrow$
 $\forall (A : \mathbf{Set})(r : \mu A), h \ A \ r = \text{Mlt } s \ r$.

provide naturality of `Mlt s`:

Section `MltNAT`.

Variable G : $\mathbf{k1}$.

Variable mG : $\mathbf{mon} \ G$.

Variable s : $\forall X : \mathbf{k1}, X \subseteq G \rightarrow F X \subseteq G$.

Variable $smGpNAT$: $\forall (X : \mathbf{k1})(it : X \subseteq G)(ef : \mathbf{EFct} X)$,
 $\mathbf{NAT} \ it \rightarrow \mathbf{NAT} \ (s \ it)$.

Instance `MltNAT` : $\mathbf{NAT} \ (\text{Mlt } s)$.

End `MltNAT`.

Section Canonization.

Variable *Fmon2br*: mon2br *F*.

Definition canonize: $\mu \subseteq \mu :=$

Mlt (fun (*X* : k1) (*it* : $X \subseteq \mu$) (*A* : Set) (*t* : $F X A$) \Rightarrow
InCan (*Fmon2br mapmu2 it t*)).

Definition OutCan: $\mu \subseteq F \mu :=$

Mlt (fun *X it A t* \Rightarrow *Fmon2br mapmu2* (fun *A* \Rightarrow InCan(*A*:= *A*) \circ (*it A*)) *t*)).

End Canonization.

End LNMITDEF.

Chapter 2

Library LNGMItPred

LNGMItPred.v Version 1.5 March 2009 does not need impredicative Set, runs under V8.2, later tested with version 8.2pl1

Copyright Ralph Matthes, I.R.I.T., C.N.R.S. & University of Toulouse

We provide basic definitions and the predicative specification of LNGMIt as an extension of LNMIIt and infer general theorems from this specification.

forms part of the code that comes with a submission to the journal Science of Computer Programming

Require Import LNMIItPred.

Require Import Utf8.

refined notion of less than on type transformations:

Definition less_k1 (X G: k1): Type :=
 $\forall (A B: \text{Set}), (A \rightarrow B) \rightarrow X A \rightarrow G B.$
less_k1 is displayed as the infix \leq .

Class mon (X: k1): Type := map: $X \leq X$.

Lemma monOk: mon = LNMIItPred.mon.

generalized refined containment:

Definition gless_k1 (X H G: k1): Type :=
 $\forall (A B: \text{Set}), (A \rightarrow H B) \rightarrow X A \rightarrow G B.$

Notation " $X <_{\{H\}} G$ " := (gless_k1 X H G) (at level 60).

Definition ext (X G: k1)(h: $X \leq G$): Prop :=
 $\forall (A B: \text{Set})(f g: A \rightarrow B),$
 $(\forall a, f a = g a) \rightarrow \forall r, h _ _ f r = h _ _ g r.$

Lemma extOk (X: k1){m: mon X}: ext m = LNMIItPred.ext(mX := m).

Definition gext (H X G: k1)(h: $X <_{\{H\}} G$): Prop :=
 $\forall (A B: \text{Set})(f g: A \rightarrow H B),$

$$(\forall a, f \ a = g \ a) \rightarrow \forall r, h \ _ _ f \ r = h \ _ _ g \ r.$$

Lemma gextlsGen: $\forall (X \ G: \mathbf{k1})(h: X \leq G)$,
 $\text{ext } h = \text{gext } h$.

Require Import Setoid.

Require Import Morphisms.

Definition gext' $(H \ X \ G: \mathbf{k1})(h: X <_{-}\{\mathbf{H}\} \ G): \text{Prop} :=$
 $\forall (A \ B: \text{Set}), \mathbf{Morphism} ((@eq \ A ==> @eq \ (H \ B)) ==> @eq \ (X \ A) ==> @eq \ (G \ B)) (h \ A \ B)$.

Lemma gextEquiv $(H \ X \ G: \mathbf{k1})(h: X <_{-}\{\mathbf{H}\} \ G): \text{gext } h \leftrightarrow \text{gext}' \ h$.

This was essentially the same proof as that for *LNMIItPred.extEquiv*.

the first part of the naturality law for inhabitants of $X \leq G$ (Definition 1/2 in the paper):

Definition nat1 $(X \ G: \mathbf{k1})(mG: \mathbf{mon} \ G)(h: X \leq G): \text{Prop} :=$
 $\forall (A \ B \ C: \text{Set})(f: A \rightarrow B)(g: B \rightarrow C)(x: X \ A),$
 $mG \ B \ C \ g \ (h \ A \ B \ f \ x) = h \ A \ C \ (g \circ f) \ x$.

the same generalized to monotone H :

Definition gnat1 $(X \ H \ G: \mathbf{k1})(mH: \mathbf{mon} \ H)(mG: \mathbf{mon} \ G)(h: X <_{-}\{\mathbf{H}\} \ G): \text{Prop} :=$
 $\forall (A \ B \ C: \text{Set})(f: A \rightarrow H \ B)(g: B \rightarrow C)(x: X \ A),$
 $mG \ B \ C \ g \ (h \ A \ B \ f \ x) = h \ A \ C \ ((mH \ _ _ g) \circ f) \ x$.

Definition monld: $\mathbf{mon} \ (\text{fun } A \Rightarrow A) := \text{fun } A \ B \ f \ x \Rightarrow f \ x$.

Lemma gnat1lsGen: $\forall (X \ G: \mathbf{k1})(mG: \mathbf{mon} \ G)(h: X \leq G)$,
 $\text{nat1 } mG \ h = \text{gnat1 } \text{monld} \ mG \ h$.

the second part of the naturality law for inhabitants of $X \leq G$ (Definition 1/2 in the paper):

Definition nat2 $(X \ G: \mathbf{k1})(mX: \mathbf{mon} \ X)(h: X \leq G): \text{Prop} :=$
 $\forall (A \ B \ C: \text{Set})(f: A \rightarrow B)(g: B \rightarrow C)(x: X \ A),$
 $h \ B \ C \ g \ (mX \ A \ B \ f \ x) = h \ A \ C \ (g \circ f) \ x$.

Definition gnat2 $(X \ H \ G: \mathbf{k1})(mX: \mathbf{mon} \ X)(h: X <_{-}\{\mathbf{H}\} \ G): \text{Prop} :=$
 $\forall (A \ B \ C: \text{Set})(f: A \rightarrow B)(g: B \rightarrow H \ C)(x: X \ A),$
 $h \ B \ C \ g \ (mX \ A \ B \ f \ x) = h \ A \ C \ (g \circ f) \ x$.

Lemma gnat2lsGen: $\forall (X \ G: \mathbf{k1})(mX: \mathbf{mon} \ X)(h: X \leq G)$,
 $\text{nat2 } mX \ h = \text{gnat2 } mX \ h$.

Definition lessTosub $(X \ G: \mathbf{k1}): X \leq G \rightarrow X \subseteq G$.

Lemma nat1nat2NAT $(X \ G: \mathbf{k1})(mX: \mathbf{mon} \ X)(mG: \mathbf{mon} \ G)(h: X \leq G):$
 $\text{nat1 } mG \ h \rightarrow \text{nat2 } mX \ h \rightarrow \mathbf{NAT}(mX := mX)(mY := mG) (\text{lessTosub } h)$.

Definition glessTosub $(X \ H \ G: \mathbf{k1}): X <_{-}\{\mathbf{H}\} \ G \rightarrow (\text{fun } A \Rightarrow X(H \ A)) \subseteq G$.

Lemma glessTosublsGen: $\forall (X \ G: \mathbf{k1})(h: X \leq G)(A: \text{Set})(t: X \ A)$,
 $\text{glessTosub } h \ A \ t = \text{lessTosub } h \ A \ t$.

The following is Lemma 2 in the paper.

Lemma `gnat1gnat2NAT` ($X H G: \mathbf{k1}$)($mX: \mathbf{mon} X$)($mH: \mathbf{mon} H$)($mG: \mathbf{mon} G$)($h: X <_{-}\{\mathbf{H}\} G$): `gnat1` $mH mG h \rightarrow$ `gnat2` $mX h \rightarrow$ **NAT** (`glessTosub` h) ($mX := \mathbf{moncomp}(mX := mX)$)($mY := mH$))($mY := mG$).

through instantiation, we can provide an alternative proof to `nat1nat2NAT`:

Instance `nat1nat2NAT_ALT`: $\forall (X G: \mathbf{k1})(mX: \mathbf{mon} X)(mG: \mathbf{mon} G)(h: X \leq G)$,
`nat1` $mG h \rightarrow$ `nat2` $mX h \rightarrow$ **NAT**($mX := mX$)($mY := mG$) (`lessTosub` h).

As a digression, we develop a partial inverse of `gnat1gnat2NAT` (extending Mac Lane's exercise):

Definition `subTogless` ($X H G: \mathbf{k1}$)($mX: \mathbf{mon} X$): (`fun` $A \Rightarrow X(H A)$) $\subseteq G \rightarrow X <_{-}\{\mathbf{H}\} G$.

Defined.

Lemma `subToglessTosub` ($X H G: \mathbf{k1}$)($mX: \mathbf{mon} X$)($f1X: \mathbf{fct1}(mX := mX)$)($g: (\mathbf{fun} A \Rightarrow X(H A)) \subseteq G$)($A: \mathbf{Set}$)($x: X(H A)$): `glessTosub` (`subTogless` $H mX g$) $A x = g A x$.

Lemma `NATgnat1` ($X H G: \mathbf{k1}$)($mX: \mathbf{mon} X$)($f2X: \mathbf{fct2}(mX := mX)$)($mH: \mathbf{mon} H$)($mG: \mathbf{mon} G$)($g: (\mathbf{fun} A \Rightarrow X(H A)) \subseteq G$): **NAT** $g (mX := \mathbf{moncomp}(mX := mX)$)($mY := mH$))($mY := mG$) \rightarrow `gnat1` $mH mG$ (`subTogless` $H mX g$).

Lemma `subToglessgnat2` ($X H G: \mathbf{k1}$)($mX: \mathbf{mon} X$)($f2X: \mathbf{fct2}(mX := mX)$)($g: (\mathbf{fun} A \Rightarrow X(H A)) \subseteq G$): `gnat2` mX (`subTogless` $H mX g$).

because of the previous lemma, the condition on `gnat2` seems unavoidable in the following:

Lemma `glessTosubTogless` ($X H G: \mathbf{k1}$)($mX: \mathbf{mon} X$)($h: X <_{-}\{\mathbf{H}\} G$)($A B: \mathbf{Set}$)($f: A \rightarrow H B$)($x: X A$): `gext` $h \rightarrow$ `gnat2` $mX h \rightarrow$ `subTogless` $H mX$ (`glessTosub` h) $B f x = h A B f x$.

end of digression

Definition `polyExtless` ($X_1 X_2 Y_1 Y_2: \mathbf{k1}$)($t: X_1 \leq X_2 \rightarrow Y_1 \leq Y_2$): **Prop** :=
 $\forall (g h: X_1 \leq X_2)(A B: \mathbf{Set})(f: A \rightarrow B)(y: Y_1 A)$,
 $(\forall (A B: \mathbf{Set})(f: A \rightarrow B)(x: X_1 A), g A B f x = h A B f x) \rightarrow$
 $t g A B f y = t h A B f y$.

Definition `polyExtgless` ($H X_1 X_2 Y_1 Y_2: \mathbf{k1}$)($t: X_1 <_{-}\{\mathbf{H}\} X_2 \rightarrow Y_1 <_{-}\{\mathbf{H}\} Y_2$): **Prop** :=
 $\forall (g h: X_1 <_{-}\{\mathbf{H}\} X_2)(A B: \mathbf{Set})(f: A \rightarrow H B)(y: Y_1 A)$,
 $(\forall (A B: \mathbf{Set})(f: A \rightarrow H B)(x: X_1 A), g A B f x = h A B f x) \rightarrow$
 $t g A B f y = t h A B f y$.

Lemma `polyExtlessOk` ($X_1 X_2 Y_1 Y_2: \mathbf{k1}$)($t: X_1 \leq X_2 \rightarrow Y_1 \leq Y_2$):
`polyExtgless` $t =$ `polyExtless` t .

Definition `polyExtgless'` ($H X_1 X_2 Y_1 Y_2: \mathbf{k1}$): ($X_1 <_{-}\{\mathbf{H}\} X_2 \rightarrow Y_1 <_{-}\{\mathbf{H}\} Y_2$) \rightarrow **Prop** :=
Morphism ((`forall_relation` (`fun` $A: \mathbf{Set} \Rightarrow$ `forall_relation` (`fun` $B: \mathbf{Set} \Rightarrow$ (**@eq** ($A \rightarrow H B$)))
 $==>$ **@eq** ($X_1 A$) $==>$ **@eq** ($X_2 B$)))) $==>$ (`forall_relation` (`fun` $A: \mathbf{Set} \Rightarrow$ `forall_relation` (`fun` $B: \mathbf{Set} \Rightarrow$ (**@eq** ($A \rightarrow H B$))) $==>$ **@eq** ($Y_1 A$) $==>$ **@eq** ($Y_2 B$))))))%signature.

Lemma polyExtglessEquiv ($H X_1 X_2 Y_1 Y_2: k1$)($t: X_1 <_{-}\{H\} X_2 \rightarrow Y_1 <_{-}\{H\} Y_2$):
polyExtgless $t \leftrightarrow$ polyExtgless' t .

for curiosity, we define a variant of polyExtgless

Definition polyExtgless'' ($H X_1 X_2 Y_1 Y_2: k1$): ($X_1 <_{-}\{H\} X_2 \rightarrow Y_1 <_{-}\{H\} Y_2$) \rightarrow Prop
:= **Morphism** ((forall_relation (fun A: Set \Rightarrow forall_relation (fun B: Set \Rightarrow (@eq A \Rightarrow
@eq (H B)) \Rightarrow @eq (X₁ A) \Rightarrow @eq (X₂ B)))) \Rightarrow (forall_relation (fun A: Set \Rightarrow
forall_relation (fun B: Set \Rightarrow (@eq A \Rightarrow @eq (H B)) \Rightarrow @eq (Y₁ A) \Rightarrow @eq (Y₂
B))))))%signature.

Module Type LNGMIT_TYPE.

Declare Module Import LNM: LNMIT_TYPE.

Definition F:= F.

Definition FpEFct:= FpEFct.

Definition $\mu_0 := \mu_0$.

Definition $\mu := \mu$.

Definition mapmu2 := mapmu2.

Definition MltType:= MltType.

Definition Mlt0 := Mlt0.

Definition Mlt := Mlt.

Definition lnType := lnType.

Definition ln := ln.

Definition mapmu2Red:= mapmu2Red.

Definition MltRed:= MltRed.

Definition mu2lndType:= mu2lndType.

Definition mu2lnd:= mu2lnd.

Parameter GMlt0: $\forall (H G: k1), (\forall X: k1, X <_{-}\{H\} G \rightarrow F X <_{-}\{H\} G) \rightarrow \mu <_{-}\{H\} G$.

Definition GMlt: $\forall (H G: k1), (\forall X: k1, X <_{-}\{H\} G \rightarrow F X <_{-}\{H\} G) \rightarrow \mu <_{-}\{H\} G$
:= fun (H G: k1) s (A: Set) B f t \Rightarrow GMlt0(H:= H)(G:= G)(A:= A) s B f t.

Axiom GMltRed : $\forall (H G: k1)(s: \forall X: k1, X <_{-}\{H\} G \rightarrow F X <_{-}\{H\} G)(A B: Set)(f: A$
 $\rightarrow H B)(X: k1)(ef: \mathbf{EFct} X)(j: X \subseteq \mu)(n: \mathbf{NAT} j)(t: F X A),$

GMlt s _ f (ln ef (j:= j) n t) =

s X (fun (A B: Set) (f: A \rightarrow H B) \Rightarrow (GMlt s _ f) \circ (j A)) A B f t.

End LNGMIT_TYPE.

prove theorems about the components of LNGMIT_TYPE

Module LNGMITDEF(M: LNGMIT_TYPE).

Import M.

Module LNMITDEF := LNMITDEF M.

Import LNMItDef.

Section GMlt.

Variables H G: k1.

Variable s : $\forall X: \mathbf{k1}, X <_{-}\{\mathbf{H}\} G \rightarrow \mathbf{F} X <_{-}\{\mathbf{H}\} G$.

the behaviour for canonical elements:

Lemma $\mathbf{GMltRedCan}$: $\forall (A B: \mathbf{Set})(f: A \rightarrow H B)(t: \mathbf{F} \mu A)$,
 $\mathbf{GMlt} s _ f (\mathbf{InCan} t) = s (\mathbf{GMlt} s) f t$.

The following is Theorem 4 in the paper.

Lemma $\mathbf{GMltUni}$: $\forall (h: \mu <_{-}\{\mathbf{H}\} G)$,
 $(\forall (X: \mathbf{k1}), \mathbf{polyExtgless}(s(X:= X))) \rightarrow$
 $(\forall (A B: \mathbf{Set})(f: A \rightarrow H B)(X: \mathbf{k1})(ef: \mathbf{EFct} X)(j: X \subseteq \mu)(n: \mathbf{NAT} j)(t: \mathbf{F} X A)$,
 $h A B f (\mathbf{In} ef n t) =$
 $s (\mathbf{fun} (A B: \mathbf{Set}) (f: A \rightarrow H B) \Rightarrow (h A B f) \circ (j A)) f t \rightarrow$
 $\forall (A B: \mathbf{Set})(f: A \rightarrow H B)(r: \mu A), h A B f r = \mathbf{GMlt} s _ f r$.

For the remainder, we require the following:

Hypothesis $spExt$: $\forall (X: \mathbf{k1})(h: X <_{-}\{\mathbf{H}\} G), \mathbf{gext} h \rightarrow \mathbf{gext} (s h)$.

Section $\mathbf{GMltsExt}$.

$\mathbf{GMlt} s$ only depends on the extension of its functional argument

The following is Theorem 5 in the paper.

Lemma $\mathbf{GMltsExt}$: $\mathbf{gext} (\mathbf{GMlt} s)$.

End $\mathbf{GMltsExt}$.

for curiosity, we can now prove the variant of $\mathbf{GMltUni}$ that uses $\mathbf{polyExtgless''}$ instead of $\mathbf{polyExtgless}$, but this is under the extra assumption $spExt$ of the enclosing section \mathbf{GMlt}

Lemma $\mathbf{GMltUni''}$: $\forall (h: \mu <_{-}\{\mathbf{H}\} G)$,
 $(\forall (X: \mathbf{k1}), \mathbf{polyExtgless''}(s(X:= X))) \rightarrow$
 $(\forall (A B: \mathbf{Set})(f: A \rightarrow H B)(X: \mathbf{k1})(ef: \mathbf{EFct} X)(j: X \subseteq \mu)(n: \mathbf{NAT} j)(t: \mathbf{F} X A)$,
 $h A B f (\mathbf{In} ef n t) =$
 $s (\mathbf{fun} (A B: \mathbf{Set}) (f: A \rightarrow H B) \Rightarrow (h A B f) \circ (j A)) f t \rightarrow$
 $\forall (A B: \mathbf{Set})(f: A \rightarrow H B)(r: \mu A), h A B f r = \mathbf{GMlt} s _ f r$.

Section $\mathbf{GMltsNat1}$.

Variable mH : $\mathbf{mon} H$.

Variable mG : $\mathbf{mon} G$.

Hypothesis $smGpgnat1$: $\forall (X: \mathbf{k1})(h: X <_{-}\{\mathbf{H}\} G), \mathbf{EFct} X \rightarrow \mathbf{gext} h \rightarrow \mathbf{gnat1} mH mG h$
 $\rightarrow \mathbf{gnat1} mH mG (s h)$.

The following is Theorem 6 in the paper.

Lemma $\mathbf{GMltsNat1}$: $\mathbf{gnat1} mH mG (\mathbf{GMlt} s)$.

End $\mathbf{GMltsNat1}$.

Section $\mathbf{GMltsNat2}$.

Hypothesis $smGpgnat2$: $\forall (X: \mathbf{k1})(h: X <_{-}\{\mathbf{H}\} G)(ef: \mathbf{EFct} X), \mathbf{gext} h \rightarrow \mathbf{gnat2} m h \rightarrow$
 $\mathbf{gnat2} m (s h)$.

this is the part that needs the built-in naturality of *LNGMIT*

The following is Theorem 7 in the paper.

Lemma GMItsNat2: $\text{gnat2 mapmu2 (GMlt } s)$.

now we may use the assertion

End GMItsNat2.

Section GMItsNAT.

Variable mH : **mon** H .

Variable mG : **mon** G .

Hypothesis $smGpgnat1$: $\forall (X: \mathbf{k1})(h: X \leftarrow \{H\} G), \mathbf{EFct} X \rightarrow \text{gext } h \rightarrow \text{gnat1 } mH \ mG \ h$
 $\rightarrow \text{gnat1 } mH \ mG \ (s \ h)$.

Hypothesis $smGpgnat2$: $\forall (X: \mathbf{k1})(h: X \leftarrow \{H\} G)(ef: \mathbf{EFct} X), \text{gext } h \rightarrow \text{gnat2 } m \ h \rightarrow$
 $\text{gnat2 } m \ (s \ h)$.

Lemma GMItsNAT: **NAT** ($\text{glessTosub (GMlt } s)$) ($mX := \text{moncomp}(mX := \text{mapmu2})(mY :=$
 $mH)$)($mY := mG$).

End GMItsNAT.

End GMlt.

End LNGMITDEF.

Chapter 3

Library LamFlatPred

LamFlatPred.v Version 1.1 January 2010 does not need impredicative Set, runs under V8.2, tested with version 8.2pl1

Copyright Ralph Matthes, I.R.I.T., C.N.R.S. & University of Toulouse

forms part of the code that comes with a submission to the journal Science of Computer Programming

Require Import LNMItpred.

Require Import LNGMItpred.

Require Import List.

Require Import Utf8.

Open Scope *type_scope*.

Definition LambF($X: k1$)($A: Set$) := $A + X A \times X A + X(\mathbf{option} A)$.

Instance lambFpEFct : **pEFct** LambF.

Definition LamF($X: k1$)($A: Set$) := LambF $X A + X (X A)$.

Instance lamFpEFct_auto : **pEFct** LamF.

The following lemmas loosely correspond to the definition of the term of type $\forall (X:k1), \mathbf{mon} X \rightarrow \mathbf{mon}(\mathbf{LamF} X)$ at the end of Section 4.1. Note the use of **idpEFct** and **idpEFct_eta** in the recursive descriptions.

Definition pvar ($X: k1$)($A: Set$)($a: A$): LamF $X A$:=
inl ($X (X A)$) (inl ($X (\mathbf{option} A)$) (inl ($X A \times X A$) a)).

Lemma pvar_m_auto ($X: k1$)($ef: \mathbf{EFct} X$)($A B: Set$)($f: A \rightarrow B$)($a: A$):
 $m (EFct:= \mathbf{lamFpEFct_auto} ef) f (\mathbf{pvar} X a) = \mathbf{pvar} X (f a)$.

Definition papp ($X: k1$)($A: Set$)($t_1 t_2: X A$): LamF $X A$:=
inl ($X (X A)$) (inl ($X (\mathbf{option} A)$) (inr $A (t_1, t_2)$)).

Lemma papp_m_auto ($X: k1$)($ef: \mathbf{EFct} X$)($A B: Set$)($f: A \rightarrow B$)($t_1 t_2: X A$):

$m (EFct := \text{lamFpEFct_auto } ef) f (\text{papp } X \ A \ t_1 \ t_2) = \text{papp } X \ B \ (m (EFct := \text{idpEFct_eta } ef) f \ t_1) (m (EFct := \text{idpEFct_eta } ef) f \ t_2).$

Definition $\text{pabs } (X: \text{k1})(A: \text{Set})(r: X \ (\text{option } A))$: $\text{LamF } X \ A :=$
 $\text{inr } (X \ (X \ A)) \ (\text{inr } (A + X \ A \times X \ A) \ r).$

Lemma $\text{pabsE_m_auto } (X: \text{k1})(ef: \mathbf{EFct} \ X)(A \ B: \text{Set})(f: A \rightarrow B)(r: X \ (\text{option } A))$:
 $m (EFct := \text{lamFpEFct_auto } ef) f (\text{pabs } X \ A \ r) = \text{pabs } X \ B \ (m (EFct := \text{idpEFct } ef) (\text{option_map } f) \ r).$

Definition $\text{pflat } (X: \text{k1})(A: \text{Set})(ee: X \ (X \ A))$: $\text{LamF } X \ A :=$
 $\text{inr } (\text{LambF } X \ A) \ ee.$

Lemma $\text{pflat_m_auto } (X: \text{k1})(ef: \mathbf{EFct} \ X)(A \ B: \text{Set})(f: A \rightarrow B)(ee: X \ (X \ A))$:
 $m (EFct := \text{lamFpEFct_auto } ef) f (\text{pflat } X \ A \ ee) = \text{pflat } X \ B \ (m (EFct := \text{idpEFct } ef) (m (EFct := \text{idpEFct_eta } ef) f) \ ee).$

the by-hand definition at the end of Section 4.1 in the paper:

Definition $\text{lamFpEFct_m } (X: \text{k1})(mX: \mathbf{mon} \ X)$: $\mathbf{mon} \ (\text{LamF } X).$

Instance lamFpEFct : $\mathbf{pEFct} \ \text{LamF}.$

the obvious consequence

Lemma $\text{lamFpEFctlamFpEFct_m } (X: \text{k1})(ef: \mathbf{EFct} \ X)$: $@m _ \ (\text{lamFpEFct } ef) = \text{lamFpEFct_m} \ (@m _ \ ef).$

some preparations for lists

`listk1` is needed because of sort polymorphism

Definition $\text{listk1 } (A: \text{Set}) : \text{Set} := \mathbf{list} \ A.$

Fixpoint $\text{filterSome } (A: \text{Type})(l: \mathbf{list}(\text{option } A))\{\text{struct } l\} : \mathbf{list} \ A :=$
 $\text{match } l \ \text{with } \text{nil} \Rightarrow \text{nil}$
 $\quad | \ \text{None} :: \text{rest} \Rightarrow \text{filterSome } \text{rest}$
 $\quad | \ \text{Some } a :: \text{rest} \Rightarrow a :: \text{filterSome } \text{rest}$
 end.

Lemma $\text{filterSomeIn } (A: \text{Type})(a: A)(l: \mathbf{list}(\text{option } A))$:
 $\text{In } (\text{Some } a) \ l \leftrightarrow \text{In } a \ (\text{filterSome } l).$

Instance listmap : $\mathbf{LNMPred.mon} \ \text{listk1}.$

Definition $\text{optionk1 } (A: \text{Set}) : \text{Set} := \mathbf{option} \ A.$

Instance filterSomeNAT : $\mathbf{NAT} \ \text{filterSome} \ (mX := \text{moncomp}(X := \text{listk1})(Y := \text{optionk1})(mX := \text{map})(mY := \text{option_map}))(Y := \text{listk1}).$

Lemma $\text{flat_mapext } (A \ B: \text{Set})(f \ g: A \rightarrow \mathbf{list} \ B)(l: \mathbf{list} \ A)$:
 $(\forall a, f \ a = g \ a) \rightarrow \text{flat_map } f \ l = \text{flat_map } g \ l.$

interaction of `flat_map` with `map`

Lemma $\text{flat_mapLaw1 } (A \ B \ C: \text{Set})(f: A \rightarrow B)(g: B \rightarrow \mathbf{list} \ C)(l: \mathbf{list} \ A)$:

$\text{flat_map } g (\text{map } f l) = \text{flat_map } (g \circ f) l.$

an instructive instance, but not needed in the sequel:

Lemma `flat_mapLaw1Inst` ($A B : \text{Set}$)($f : A \rightarrow \text{list } B$)($l : \text{list } A$):

$\text{flat_map } (\text{id}(A := \text{list } B)) (\text{map } f l) = \text{flat_map } f l.$

Lemma `flat_mapLaw2` ($A B C : \text{Set}$) ($f : A \rightarrow \text{list } B$) ($g : B \rightarrow C$) ($l : \text{list } A$):

$\text{map } g (\text{flat_map } f l) = \text{flat_map } (\text{map } g \circ f) l.$

we use `moncomp` from *LNMItpred.v*

Lemma `flat_mapNAT` ($X : \text{k1}$)($mX : \text{mon } X$)($h : X \subseteq \text{listk1}$)($n : \text{NAT } h (mX := mX) (mY := \text{map})$): $\text{NAT}(Y := \text{listk1}) (\text{fun } A \Rightarrow \text{flat_map } (h A)) (mX := \text{moncomp}(X := \text{listk1})(mX := \text{map}))(mY := mX) (mY := \text{map}).$

Module Type `LAM` := `LNGMIT_TYPE` with Definition `LNMF := LamF`

with Definition `LNMFpEFct := lamFpEFct.`

Module `LAMFLAT` (*LamBase*: `LAM`).

Module `LAMM` := `LNGMITDEF LAMBASE.`

Module `LAMMMITDEF` := `LAMM.LNMITDEF.`

Definition `Lam` : `k1` := `LamBase.mu2.`

the canonical datatype constructors

Definition `var` ($A : \text{Set}$)($a : A$): `Lam A` :=
`LamMMItDef.lnCan (inl _ (inl _ (inl _ a)))`.

Definition `app` ($A : \text{Set}$)($t_1 t_2 : \text{Lam } A$): `Lam A` :=
`LamMMItDef.lnCan (inl _ (inl _ (inr _ (t_1, t_2))))`.

there is a conflict with *List.app*!

Lemma `app_cong`($A : \text{Set}$)($s_1 s_2 t_1 t_2 : \text{Lam } A$):

$s_1 = t_1 \rightarrow s_2 = t_2 \rightarrow \text{app } s_1 s_2 = \text{app } t_1 t_2.$

Definition `abs` ($A : \text{Set}$)($r : \text{Lam}(\text{option } A)$): `Lam A` := `LamMMItDef.lnCan (inl _ (inr _ r))`.

Definition `flat` ($A : \text{Set}$)($ee : \text{Lam}(\text{Lam } A)$): `Lam A` := `LamMMItDef.lnCan (inr _ ee)`.

lists of free variables of lambda terms, just with plain Mendler iteration see Section 2.2 in the paper

Definition `FV` : `Lam` \subseteq `listk1`.

Definition `sFV` := `fun` ($X : \text{k1}$) ($it : X \subseteq \text{listk1}$) ($A : \text{Set}$) ($t : \text{LamF } X A$) \Rightarrow

`match t with`

| `inl (inl (inl a))` $\Rightarrow a :: \text{nil}$

| `inl (inl (inr (pair t_1 t_2)))` $\Rightarrow it A t_1 ++ it A t_2$

| `inl (inr r)` $\Rightarrow \text{filterSome } (it (\text{option } A) r)$

| `inr e` $\Rightarrow \text{flat_map } (it A) (it (X A) e)$

`end.`

Lemma sFV_ok: FV = LamBase.Mlt sFV.

Lemma FV_var (A: Set)(a: A): FV(var a) = a :: nil.

Lemma FV_app (A: Set)(t₁ t₂: Lam A): FV(app t₁ t₂) = FV t₁ ++ FV t₂.

Lemma FV_abs (A: Set)(r: Lam (option A)): FV(abs r) = filterSome (FV r).

Lemma FV_flat (A: Set)(ee: Lam (Lam A)):

FV(flat ee) = flat_map (FV(A:= A)) (FV ee).

renaming, see Section 2.2 in the paper

Instance lam : LNMLtPred.mon Lam.

behaviour of lam on canonical elements

Lemma lam_var (A B: Set)(f: A → B)(a: A) : lam f (var a) = var (f a).

Lemma lam_app (A B: Set)(f: A → B)(t₁ t₂: Lam A):

lam f (app t₁ t₂) = app (lam f t₁)(lam f t₂).

Lemma lam_abs (A B: Set)(f: A → B)(r: Lam (option A)):

lam f (abs r) = abs (lam (option_map f) r).

Lemma lam_flat (A B: Set)(f: A → B)(ee: Lam (Lam A)):

lam f (flat ee) = flat (lam (lam f) ee).

The "interesting question" at the end of Section 2.2 of the paper:

Instance FVNAT: NAT FV.

how renaming works on the list of free variables:

Corollary FV_ok (A B: Set)(f: A → B)(t: Lam A): FV(lam f t) = map f (FV t).

towards substitution, see Section 2.3 of the paper

Implicit Arguments Some [[A]].

Definition lift (A B: Set)(f: A → Lam B)(x: option A): Lam(option B) :=

match x with

| None ⇒ var None

| Some a ⇒ lam Some (f a)

end.

Definition subst (A B: Set)(f: A → Lam B)(t: Lam A): Lam B.

Definition s_subst :=

fun (X : k1) (it : X <-{ Lam } Lam) (A B : Set) (f : A → Lam B)

(t : LamF X A) ⇒

match t with

| inl (inl (inl a)) ⇒ f a

| inl (inl (inr (pair t₁ t₂))) ⇒ app (it A B f t₁) (it A B f t₂)

| inl (inr r) ⇒ abs (it (option A) (option B) (lift f) r)

| inr ee ⇒ flat (it (X A) (Lam B) (var (A:= Lam B) ∘ it A B f) ee)

end.

Lemma s_subst_ok: LamBase.GMlt s_subst = subst.

Lemma subst_var (A B: Set)(f: A → Lam B)(a: A):
subst f (var a) = f a.

Lemma substMonad1 (A B: Set)(f: A → Lam B)(a: A): subst f (var a) = f a.

Lemma subst_app (A B: Set)(f: A → Lam B)(t₁ t₂: Lam A):
subst f (app t₁ t₂) = app (subst f t₁)(subst f t₂).

Lemma subst_abs (A B: Set)(f: A → Lam B)(r: Lam (option A)):
subst f (abs r) = abs (subst (lift f) r).

Lemma subst_flat (A B: Set)(f: A → Lam B)(ee: Lam(Lam A)):
subst f (flat ee) = flat (subst (var(A:= Lam B) ∘ (subst f)) ee).

we would have liked to see `subst f (flat ee) = flat (lam (subst f) ee)`, but this would need a full second monad law

the generic properties of renaming:

Lemma lamext: LNMLtPred.ext(mX:= lam).

Lemma lamfct1: fct1(mX:= lam).

Lemma lamfct2 : fct2(mX:= lam).

show that the definition of `subst` qualifies for Theorem 5 in the paper

Lemma substpGext: ∀ (X : k1) (h : X <-{ Lam } Lam), gext h → gext (s_subst h).

the first item in Theorem 1 in the paper:

Lemma substext (A B: Set)(f g: A → Lam B)(t: Lam A):
(∀ a, f a = g a) → subst f t = subst g t.

Lemma FV_var_gen (X : k1)(ef : EFct X)(j : X ⊆ Lam)(n : NAT j)(A : Set)(a : A):
FV (LamBase.In ef n (pvar X a)) = a :: nil.

Lemma FV_app_gen (X : k1)(ef : EFct X)(j : X ⊆ Lam)(n : NAT j)(A : Set)(t₁ t₂: X A):
FV (LamBase.In ef n (papp X A t₁ t₂)) = FV (j A t₁) ++ FV (j A t₂).

Lemma FV_abs_gen (X : k1)(ef : EFct X)(j : X ⊆ Lam)(n : NAT j)(A : Set)(r: X (option A)):
FV (LamBase.In ef n (pabs X A r)) = filterSome (FV (j (option A) r)).

Lemma FV_flat_gen (X : k1)(ef : EFct X)(j : X ⊆ Lam)(n : NAT j)(A : Set)(ee: X(X A)):
FV (LamBase.In ef n (pflat X A ee)) = flat_map (FV(A:= A) ∘ (j A)) (FV (j (X A) ee)).

Definition occursFreeln(A: Set)(a: A)(t: Lam A): Prop := In a (FV t).

Infix "occ" := occursFreeln (at level 90).

Lemma occ_var (X : k1)(ef : EFct X)(j : X ⊆ Lam)(n : NAT j)(A : Set)(a : A):
a occ (LamBase.In ef n (pvar X a)).

Lemma `occ_appl` ($X : \mathbf{k1}$)($ef : \mathbf{EFct} X$)($j : X \subseteq \mathbf{Lam}$)($n : \mathbf{NAT} j$)($A : \mathbf{Set}$)($a : A$)($t_1 t_2 : X A$): ($a \text{ occ } (j A t_1)$) $\rightarrow a \text{ occ } (\mathbf{LamBase.In} ef n (\mathbf{papp} X A t_1 t_2))$).

Lemma `occ_appr` ($X : \mathbf{k1}$)($ef : \mathbf{EFct} X$)($j : X \subseteq \mathbf{Lam}$)($n : \mathbf{NAT} j$)($A : \mathbf{Set}$)($a : A$)($t_1 t_2 : X A$): ($a \text{ occ } (j A t_2)$) $\rightarrow a \text{ occ } (\mathbf{LamBase.In} ef n (\mathbf{papp} X A t_1 t_2))$).

Lemma `occ_abs` ($X : \mathbf{k1}$)($ef : \mathbf{EFct} X$)($j : X \subseteq \mathbf{Lam}$)($n : \mathbf{NAT} j$)($A : \mathbf{Set}$)($a : A$)($r : X (\mathbf{option} A)$): ($(\mathbf{Some} a) \text{ occ } (j (\mathbf{option} A) r)$) $\rightarrow a \text{ occ } (\mathbf{LamBase.In} ef n (\mathbf{pabs} X A r))$).

Lemma `occ_flat` ($X : \mathbf{k1}$)($ef : \mathbf{EFct} X$)($j : X \subseteq \mathbf{Lam}$)($n : \mathbf{NAT} j$)($A : \mathbf{Set}$)($a : X A$)($ee : X(X A)$):

`occursFreeln` $a (j (X A) ee) \rightarrow$
 $\forall a_0 : A, \text{occursFreeln } a_0 (j A a) \rightarrow$
 $\text{occursFreeln } a_0 (\mathbf{LamBase.In} ef n (\mathbf{pflat} X A ee))$).

now the interpretation for canonical terms although not needed for Theorem 1:

Lemma `occCan_var` ($A : \mathbf{Set}$)($a : A$): $a \text{ occ } (\mathbf{var} a)$.

Lemma `occCan_appl` ($A : \mathbf{Set}$)($a : A$)($t_1 t_2 : \mathbf{Lam} A$):
 $a \text{ occ } t_1 \rightarrow a \text{ occ } (\mathbf{app} t_1 t_2)$.

Lemma `occCan_appr` ($A : \mathbf{Set}$)($a : A$)($t_1 t_2 : \mathbf{Lam} A$):
 $a \text{ occ } t_2 \rightarrow a \text{ occ } (\mathbf{app} t_1 t_2)$.

Lemma `occCan_abs` ($A : \mathbf{Set}$)($a : A$)($r : \mathbf{Lam}(\mathbf{option} A)$):
 $(\mathbf{Some} a) \text{ occ } r \rightarrow a \text{ occ } (\mathbf{abs} r)$.

Lemma `occCan_flat` ($A : \mathbf{Set}$)($a : A$)($t : \mathbf{Lam} A$)($ee : \mathbf{Lam}(\mathbf{Lam} A)$):
 $a \text{ occ } t \rightarrow t \text{ occ } ee \rightarrow a \text{ occ } (\mathbf{flat} ee)$.

the second item in Theorem 1 in the paper:

Lemma `substext'` ($A B : \mathbf{Set}$)($f g : A \rightarrow \mathbf{Lam} B$)($t : \mathbf{Lam} A$):
 $(\forall a, a \text{ occ } t \rightarrow f a = g a) \rightarrow \mathbf{subst} f t = \mathbf{subst} g t$.

the third item in Theorem 1 in the paper needs one extra lemma:

Lemma `lift_map` ($A B C : \mathbf{Set}$)($a : \mathbf{option} A$)($f : A \rightarrow \mathbf{Lam} B$)($g : B \rightarrow C$):
 $(\mathbf{lam} (\mathbf{option_map} g) \circ \mathbf{lift} f) a = \mathbf{lift} (\mathbf{lam} g \circ f) a$.

the third item in Theorem 1 in the paper:

Lemma `substGnat1` ($A B C : \mathbf{Set}$) ($f : A \rightarrow \mathbf{Lam} B$) ($g : B \rightarrow C$) ($t : \mathbf{Lam} A$):
 $\mathbf{lam} g (\mathbf{subst} f t) = \mathbf{subst} (\mathbf{lam} g \circ f) t$.

the fourth item in Theorem 1 in the paper:

Lemma `substGnat2` ($A B C : \mathbf{Set}$)($f : A \rightarrow B$)($g : B \rightarrow \mathbf{Lam} C$)($t : \mathbf{Lam} A$):
 $\mathbf{subst} g (\mathbf{lam} f t) = \mathbf{subst} (g \circ f) t$.

just an instance

Lemma `substLaw` ($A B : \mathbf{Set}$)($f : A \rightarrow \mathbf{Lam} B$)($t : \mathbf{Lam} A$):
 $\mathbf{subst} \mathbf{id} (\mathbf{lam} f t) = \mathbf{subst} f t$.

a more elaborate formulation of naturality

Instance substNAT ($X: \mathbf{k1}$)($mX: \mathbf{mon} X$)($h: X \subseteq \mathbf{Lam}$)($n: \mathbf{NAT}(mX:= mX) h$):
 $\mathbf{NAT} (\text{fun } A \Rightarrow \text{subst } (h A)) (mX:= \text{moncomp}(mX:= \text{lam}))(mY:= mX)$.

Corollary substNATCor ($X: \mathbf{k1}$)($mX: \mathbf{mon} X$)($h: X \subseteq \mathbf{Lam}$)($n: \mathbf{NAT}(mX:= mX) h$)
 $(A B: \mathbf{Set}) (f: A \rightarrow B)(t: \mathbf{Lam} (X A))$:
 $\text{subst } (h B) (\text{lam } (mX A B f) t) = \text{lam } f (\text{subst } (h A) t)$.

Definition flatimpl: $\forall (A: \mathbf{Set}), \mathbf{Lam}(\mathbf{Lam} A) \rightarrow \mathbf{Lam} A := \text{glessTosub subst}$.

Instance flatimplNAT: $\mathbf{NAT} \text{ flatimpl } (mX:= \text{moncomp}(mX:= \text{lam}))(mY:= \text{lam})$.

the fifth item in Theorem 1 in the paper:

Lemma substMonad3 ($A B C: \mathbf{Set}$)($f: A \rightarrow \mathbf{Lam} B$)($g: B \rightarrow \mathbf{Lam} C$)($t: \mathbf{Lam} A$):
 $\text{subst } g (\text{subst } f t) = \text{subst } ((\text{subst } g) \circ f) t$.

Section 5 of the paper

preparations for the formula for the list of free variables of $\text{subst } f t$

Lemma filterSomeAppend ($A: \mathbf{Type}$)($l_1 l_2: \mathbf{list}(\mathbf{option} A)$):
 $\text{filterSome } (l_1 ++ l_2) = \text{filterSome } l_1 ++ \text{filterSome } l_2$.

Lemma filterSomeMapSome ($A: \mathbf{Type}$)($l: \mathbf{list} A$):
 $\text{filterSome } (\text{map } (\text{Some}(A:= A)) l) = l$.

Lemma filterSomeFV ($A: \mathbf{Set}$)($t: \mathbf{Lam} A$):
 $\text{filterSome } (\text{FV } (\text{lam } \text{Some } t)) = \text{FV } t$.
 this needed naturality of FV

Lemma FVsubst_aux ($A B: \mathbf{Set}$)($f: A \rightarrow \mathbf{Lam} B$)($t: \mathbf{list}(\mathbf{option} A)$):
 $\text{filterSome}(\text{flat_map}(\text{FV}(A:= \mathbf{option} B) \circ \text{lift } f) t) = \text{flat_map } (\text{FV}(A:= B) \circ f)(\text{filterSome } t)$.

this needed naturality of FV through filterSomeFV

further preparations for the last item of Theorem 1 in the paper:

Lemma flat_map_app ($A B: \mathbf{Set}$)($f: A \rightarrow \mathbf{list} B$)($l_1 l_2: \mathbf{list} A$):
 $\text{flat_map } f (l_1 ++ l_2) = \text{flat_map } f l_1 ++ \text{flat_map } f l_2$.

Lemma flat_mapMonad3 ($A B C: \mathbf{Set}$) ($f: A \rightarrow \mathbf{list} B$) ($g: B \rightarrow \mathbf{list} C$) ($l: \mathbf{list} A$):
 $\text{flat_map } g (\text{flat_map } f l) = \text{flat_map } ((\text{flat_map } g) \circ f) l$.

the last item of Theorem 1 in the paper

Lemma FVsubst ($A B: \mathbf{Set}$)($f: A \rightarrow \mathbf{Lam} B$)($t: \mathbf{Lam} A$):
 $\text{FV } (\text{subst } f t) = \text{flat_map } (\text{FV}(A:= B) \circ f) (\text{FV } t)$.
 here enters naturality of FV

its consequence close to the end of Section 2.3 of the paper:

Lemma subst_occ ($A B: \mathbf{Set}$)($f: A \rightarrow \mathbf{Lam} B$)($t: \mathbf{Lam} A$)($b: B$):
 $b \text{ occ } (\text{subst } f t) \rightarrow \exists a: A, (a \text{ occ } t) \wedge (b \text{ occ } f a)$.

hereditarily canonical terms

Definition 3 in the paper

Inductive **can** : $\forall (A: \text{Set}), \text{Lam } A \rightarrow \text{Prop} :=$
 | **can_var** : $\forall (A: \text{Set})(a: A), \text{can } (\text{var } a)$
 | **can_app** : $\forall (A: \text{Set})(t_1 t_2: \text{Lam } A), \text{can } t_1 \rightarrow \text{can } t_2 \rightarrow \text{can}(\text{app } t_1 t_2)$
 | **can_abs** : $\forall (A: \text{Set})(r: \text{Lam}(\text{option } A)), \text{can } r \rightarrow \text{can } (\text{abs } r)$
 | **can_flat**: $\forall (A: \text{Set})(ee: \text{Lam}(\text{Lam } A)),$
 can $ee \rightarrow (\forall t: \text{Lam } A, t \text{ occ } ee \rightarrow \text{can } t) \rightarrow \text{can } (\text{flat } ee).$

Here comes material that did not form part of the original submission of the SCP paper. It was stimulated by a question of one of the referees how **can** is related to what will be called *LamP* below. Many thanks to the anonymous referee for that. It was easy for me to answer that question since I had already studied binary versions of those unary predicates before. However, those developments would have led too far from the main issue of the paper.

Definition **optionpred** ($A: \text{Type}$)($P: A \rightarrow \text{Prop}$): **option** $A \rightarrow \text{Prop}.$

an axiomatic definition of a truly nested inductive predicate on **Lam**:

Parameter *LamP*: $\forall A: \text{Set}, (A \rightarrow \text{Prop}) \rightarrow \text{Lam } A \rightarrow \text{Prop}.$
 Axiom *LamP_var*: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(a: A), P a \rightarrow \text{LamP } P (\text{var } a).$
 Axiom *LamP_app*: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(t_1 t_2: \text{Lam } A), \text{LamP } P t_1 \rightarrow \text{LamP } P t_2 \rightarrow \text{LamP } P (\text{app } t_1 t_2).$
 Axiom *LamP_abs*: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(r: \text{Lam}(\text{option } A)), \text{LamP } (\text{optionpred } P) r \rightarrow \text{LamP } P (\text{abs } r).$
 Axiom *LamP_flat*: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(ee: \text{Lam}(\text{Lam } A)), \text{LamP } (\text{LamP } P) ee \rightarrow \text{LamP } P (\text{flat } ee).$
 Axiom *LamP_ind*: $\forall (Q: \forall A: \text{Set}, (A \rightarrow \text{Prop}) \rightarrow \text{Lam } A \rightarrow \text{Prop}),$
 $(\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(a: A), P a \rightarrow Q A P (\text{var } a))$
 $\rightarrow (\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(t_1 t_2: \text{Lam } A), \text{LamP } P t_1 \rightarrow Q A P t_1 \rightarrow \text{LamP } P t_2 \rightarrow Q A P t_2 \rightarrow Q A P (\text{app } t_1 t_2))$
 $\rightarrow (\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(r: \text{Lam}(\text{option } A)), \text{LamP } (\text{optionpred } P) r \rightarrow Q (\text{option } A) (\text{optionpred } P) r \rightarrow Q A P (\text{abs } r))$
 $\rightarrow (\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(ee: \text{Lam}(\text{Lam } A)), \text{LamP } (\text{LamP } P) ee \rightarrow Q (\text{Lam } A) (Q A P) ee \rightarrow Q A P (\text{flat } ee))$
 $\rightarrow \forall (A: \text{Set})(P: A \rightarrow \text{Prop})(t: \text{Lam } A), \text{LamP } P t \rightarrow Q A P t.$

the last clause is in plain analogy with the other ones

Definition **univpred** ($A: \text{Type}$): $A \rightarrow \text{Prop}.$

a variant that is accepted as inductive definition by Coq:

Inductive **LamP'** : $\forall (A: \text{Set}), (A \rightarrow \text{Prop}) \rightarrow \text{Lam } A \rightarrow \text{Prop} :=$
 | **LamP'_var**: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(a: A), P a \rightarrow \text{LamP}' P (\text{var } a)$
 | **LamP'_app**: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(t_1 t_2: \text{Lam } A), \text{LamP}' P t_1 \rightarrow \text{LamP}' P t_2 \rightarrow \text{LamP}' P (\text{app } t_1 t_2)$
 | **LamP'_abs**: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(r: \text{Lam}(\text{option } A)), \text{LamP}' (\text{optionpred } P) r \rightarrow \text{LamP}' P (\text{abs } r)$

| **LamP'_flat**: $\forall (A: \text{Set})(P: A \rightarrow \text{Prop})(ee: \text{Lam}(\text{Lam } A)), \mathbf{LamP}' (@\text{univpred } (\text{Lam } A)) ee \rightarrow (\forall t: \text{Lam } A, t \text{ occ } ee \rightarrow \mathbf{LamP}' P t) \rightarrow \mathbf{LamP}' P (\text{flat } ee)$.

Lemma **LamP'ImpCan** $(A: \text{Set})(P: A \rightarrow \text{Prop})(t: \text{Lam } A): \mathbf{LamP}' P t \rightarrow \mathbf{can } t$.

Require Import Setoid.

Require Import Morphisms.

Definition **subpredicate** $(A: \text{Type})(P P': A \rightarrow \text{Prop}) := \forall a:A, P a \rightarrow P' a$.

Add *Parametric Morphism* $(A: \text{Set}): (\mathbf{LamP}'(A:=A))$

with *signature* $(\text{subpredicate}(A:=A)) ==> (\text{subpredicate}(A:=\text{Lam } A))$

as *LamP'_monM*.

Lemma **CanImpLamP'univ** $(A: \text{Set})(t: \text{Lam } A): \mathbf{can } t \rightarrow \mathbf{LamP}' (@\text{univpred } A) t$.

thus, the relation between **can** and **LamP'** is rather trivial

Lemma **occ_var_inv** $(A: \text{Set})(a b: A): a \text{ occ } (\text{var } b) \rightarrow a = b$.

Lemma **occ_app_inv** $(A: \text{Set})(a: A)(t_1 t_2: \text{Lam } A): a \text{ occ } (\text{app } t_1 t_2) \rightarrow (a \text{ occ } t_1) \vee (a \text{ occ } t_2)$.

Lemma **occ_abs_inv** $(A: \text{Set})(a: A)(r: \text{Lam } (\mathbf{option } A)):$
 $a \text{ occ } (\text{abs } r) \rightarrow \text{Some } a \text{ occ } r$.

Lemma **occ_flat_inv** $(A: \text{Set})(a: A)(ee: \text{Lam } (\text{Lam } A)):$
 $a \text{ occ } (\text{flat } ee) \rightarrow \exists t: \text{Lam } A, (a \text{ occ } t) \wedge (t \text{ occ } ee)$.

an auxiliary lemma

Lemma **LamP'_FV** $(A: \text{Set})(P: A \rightarrow \text{Prop})(t: \text{Lam } A)(a: A): \mathbf{LamP}' P t \rightarrow a \text{ occ } t \rightarrow P a$.

for curiosity, an analogous lemma for *LamP*:

Lemma **LamP_FV** $(A: \text{Set})(P: A \rightarrow \text{Prop})(t: \text{Lam } A)(a: A): \text{LamP } P t \rightarrow a \text{ occ } t \rightarrow P a$.

LamP' satisfies the last closure rule of *LamP*:

Lemma **LamP'_flat'** $(A: \text{Set})(P: A \rightarrow \text{Prop})(ee: \text{Lam}(\text{Lam } A)): \mathbf{LamP}' (\mathbf{LamP}' P) ee \rightarrow \mathbf{LamP}' P (\text{flat } ee)$.

Lemma **LamPImpLamP'** $(A: \text{Set})(P: A \rightarrow \text{Prop})(t: \text{Lam } A): \text{LamP } P t \rightarrow \mathbf{LamP}' P t$.

another auxiliary lemma:

Lemma **CanImpLamP** $(A: \text{Set})(t: \text{Lam } A): \mathbf{can } t \rightarrow \forall P: A \rightarrow \text{Prop}, (\forall a: A, a \text{ occ } t \rightarrow P a) \rightarrow \text{LamP } P t$.

LamP satisfies the last closure rule of **LamP'**:

Lemma **LamP_flat'** $(A: \text{Set})(P: A \rightarrow \text{Prop})(ee: \text{Lam}(\text{Lam } A)): \text{LamP } (@\text{univpred } (\text{Lam } A)) ee \rightarrow (\forall t: \text{Lam } A, t \text{ occ } ee \rightarrow \text{LamP } P t) \rightarrow \text{LamP } P (\text{flat } ee)$.

Lemma **LamP'ImpLamP** $(A: \text{Set})(P: A \rightarrow \text{Prop})(t: \text{Lam } A): \mathbf{LamP}' P t \rightarrow \text{LamP } P t$.

Section 5.1 of the paper

we are only able to prove a weakened version of the second monad law, namely Lemma 8 in the paper:

Lemma `substMonad2` ($A: \text{Set}$)($t: \text{Lam } A$): `can` $t \rightarrow \text{subst} (\text{var}(A:= A)) t = t$.

Lemma `lamext_refined` ($A B: \text{Set}$)($f g: A \rightarrow B$)($t: \text{Lam } A$):
`can` $t \rightarrow (\forall a, a \text{ occ } t \rightarrow f a = g a) \rightarrow \text{lam } f t = \text{lam } g t$.

Lemma `lam_occ` ($A B: \text{Set}$)($f: A \rightarrow B$)($t: \text{Lam } A$)($b: B$):
 $b \text{ occ } (\text{lam } f t) \rightarrow \exists a: A, (a \text{ occ } t) \wedge (b = f a)$.

Lemma `lam_can` ($A B: \text{Set}$)($f: A \rightarrow B$)($t: \text{Lam } A$): `can` $t \rightarrow \text{can} (\text{lam } f t)$.

Lemma `subst_can` ($A B: \text{Set}$)($f: A \rightarrow \text{Lam } B$)($t: \text{Lam } A$):
 $(\forall a: A, a \text{ occ } t \rightarrow \text{can} (f a)) \rightarrow \text{can } t \rightarrow \text{can} (\text{subst } f t)$.

Lemma `lam_is_subst` ($A B: \text{Set}$)($f: A \rightarrow B$)($t: \text{Lam } A$):
`can` $t \rightarrow \text{lam } f t = \text{subst} (\text{var}(A:= B) \circ f) t$.

finally, we obtain the desired rewrite rule for `subst` in the case `flat`, but only for canonical elements

Lemma `subst_flat'` ($A B: \text{Set}$)($f: A \rightarrow \text{Lam } B$)($ee: \text{Lam}(\text{Lam } A)$):
`can` $ee \rightarrow \text{subst } f (\text{flat } ee) = \text{flat} (\text{lam} (\text{subst } f) ee)$.

Section 5.2 of the paper

Definition `Lam'` ($A: \text{Set}$): `Set` := $\{t: \text{Lam } A \mid \text{can } t\}$.

Inductive `Lam'_ALT` ($A: \text{Set}$): `Set` :=
`existLam'`: $\forall t: \text{Lam } A, \text{can } t \rightarrow \text{Lam}'_ALT A$.

Definition `Lam'_ALT_Lam'` ($A: \text{Set}$): `Lam'_ALT` $A \rightarrow \text{Lam}' A$.

Definition `Lam'_Lam'_ALT` ($A: \text{Set}$): `Lam}' A \rightarrow \text{Lam}'_ALT A.`

Lemma `Lam'_ALT_bij_Lam'` ($A: \text{Set}$):
 $(\forall t: \text{Lam}' A, \text{Lam}'_ALT_Lam' (\text{Lam}'_Lam'_ALT t) = t) \wedge$
 $(\forall t: \text{Lam}'_ALT A, \text{Lam}'_Lam'_ALT (\text{Lam}'_ALT_Lam' t) = t)$.

Definition `pi1`: `Lam}' \subseteq Lam`.

Implicit Arguments `pi1` $[[A]]$.

Definition `pi1_ALT`: `Lam}'_ALT \subseteq Lam`.

Definition `var'` ($A: \text{Set}$)($a: A$): `Lam}' A`.

Definition `app'` ($A: \text{Set}$)($t_1 t_2: \text{Lam}' A$): `Lam}' A`.

Definition `abs'` ($A: \text{Set}$)($r: \text{Lam}' (\text{option } A)$): `Lam}' A`.

Definition `flat'` ($A: \text{Set}$)($ee: \text{Lam}'(\text{Lam}' A)$): `Lam}' A`.

unfortunately, this definition has to do something on terms, namely renaming using `pi1`

Lemma `flat'_ok` ($A: \text{Set}$)($ee: \text{Lam}'(\text{Lam}' A)$):
`pi1` $(\text{flat}' ee) = \text{flat}(\text{lam } \text{pi1} (\text{pi1 } ee))$.

Instance lam': **LNMItpred.mon** Lam'.

Lemma lam'_ok (A B: Set)(f: A → B)(t: Lam' A):
pil (lam' f t) = lam f (pil t).

Definition subst' (A B: Set)(f: A → Lam' B)(t: Lam' A): Lam' B.

Lemma subst'_ok (A B: Set)(f: A → Lam' B)(t: Lam' A):
pil (subst' f t) = subst (pil ∘ f) (pil t).

Definition FV': Lam' ⊆ listk1.

Instance FV'NAT: **NAT** FV'.

Corollary FV'_ok (A B: Set)(f: A → B)(t: Lam' A):
FV'(lam' f t) = map f (FV' t).

Lemma FV'_var (A: Set)(a: A): FV'(var' a) = a :: nil.

Lemma FV'_app (A: Set)(t1 t2: Lam' A): FV'(app' t1 t2) = FV' t1 ++ FV' t2.

Lemma FV'_abs (A: Set)(r: Lam' (option A)): FV'(abs' r) = filterSome (FV' r).

Lemma FV'_flat (A: Set)(ee: Lam' (Lam' A)):
FV'(flat' ee) = flat_map (FV'(A:= A)) (FV' ee).

item 6 of Theorem 1 for Lam' in place of Lam:

Lemma FV'subst' (A B: Set)(f: A → Lam' B)(t: Lam' A):
FV'(subst' f t) = flat_map (FV'(A:= B) ∘ f) (FV' t).

Definition occursFreeln' (A: Set)(a: A)(t: Lam' A): Prop := ln a (FV' t).

Infix "occ'" := occursFreeln' (at level 90).

Lemma lam'_occ (A B: Set)(f: A → B)(t: Lam' A)(b: B):
b occ' (lam' f t) → ∃ a: A, (a occ' t) ∧ (b = f a).

Lemma lam'_occ_ALT (A B: Set)(f: A → B)(t: Lam' A)(b: B):
b occ' (lam' f t) → ∃ a: A, (a occ' t) ∧ (b = f a).

Lemma subst'_occ (A B: Set)(f: A → Lam' B)(t: Lam' A)(b: B):
b occ' (subst' f t) → ∃ a: A, (a occ' t) ∧ (b occ' f a).

Lemma subst'_occ_ALT (A B: Set)(f: A → Lam' B)(t: Lam' A)(b: B):
b occ' (subst' f t) → ∃ a: A, (a occ' t) ∧ (b occ' f a).

Definition EqLam' (A: Set)(t1 t2: Lam' A): Prop := pil t1 = pil t2.

EqLam' is displayed as the infix ≡.

Lemma EqLam'_refl (A: Set)(t: Lam' A): t ≡ t.

Lemma EqLam'_sym (A: Set)(t1 t2: Lam' A): t1 ≡ t2 → t2 ≡ t1.

Lemma EqLam'_trans (A: Set)(t1 t2 t3: Lam' A): t1 ≡ t2 → t2 ≡ t3 → t1 ≡ t3.

Add *Parametric Relation* (A: Set): (Lam' A) (EqLam'(A:= A))
reflexivity proved by (EqLam'_refl(A:= A))

symmetry *proved* by (EqLam'_sym(A:= A))
 transitivity *proved* by (EqLam'_trans(A:= A))
 as *bisimilarRel*.

we do not impose the following axiomatically

Definition proof_irrelevance := $\forall (P: \text{Prop}) (p_1 p_2: P), p_1 = p_2$.

Definition Lam'pirrProp := $\forall (A: \text{Set})(t_1 t_2: \text{Lam}' A), t_1 \equiv t_2 \rightarrow t_1 = t_2$.

Lemma Lam'pirrFromPirr: proof_irrelevance \rightarrow Lam'pirrProp.

Theorem 9 in the paper

item 1:

Lemma subst'ext' (A B: Set)(f g: A \rightarrow Lam' B)(t: Lam' A):
 $(\forall a, a \text{ occ}' t \rightarrow f a \equiv g a) \rightarrow \text{subst}' f t \equiv \text{subst}' g t$.

Corollary subst'ext (A B: Set)(f g: A \rightarrow Lam' B)(t: Lam' A):
 $(\forall a, f a \equiv g a) \rightarrow \text{subst}' f t \equiv \text{subst}' g t$.

item 2:

Lemma subst'Gnat1 (A B C : Set) (f : A \rightarrow Lam' B) (g : B \rightarrow C) (t : Lam' A):
 $\text{lam}' g (\text{subst}' f t) \equiv \text{subst}' (\text{lam}' g \circ f) t$.

item 3:

Lemma subst'Gnat2 (A B C: Set)(f: A \rightarrow B)(g: B \rightarrow Lam' C)(t: Lam' A):
 $\text{subst}' g (\text{lam}' f t) \equiv \text{subst}' (g \circ f) t$.

item 4:

Definition monad3Lam': Prop :=
 $\forall (A B C : \text{Set}) (f : A \rightarrow \text{Lam}' B) (g : B \rightarrow \text{Lam}' C) (t : \text{Lam}' A),$
 $\text{subst}' g (\text{subst}' f t) \equiv \text{subst}' (\text{subst}' g \circ f) t$.

Lemma subst'Monad3: monad3Lam'.

the lemmas that no longer need an explicit relativization to hereditarily canonical elements

item 5:

Lemma subst'Monad2 (A: Set)(t: Lam' A): $\text{subst}' (\text{var}'(A:= A)) t \equiv t$.

item 6:

Lemma lam'_is_subst' (A B: Set)(f: A \rightarrow B)(t: Lam' A):
 $\text{lam}' f t \equiv \text{subst}' (\text{var}'(A:= B) \circ f) t$.

item 7:

Lemma subst'_flat' (A B: Set)(f: A \rightarrow Lam' B)(ee: Lam'(Lam' A)):
 $\text{subst}' f (\text{flat}' ee) \equiv \text{flat}' (\text{lam}' (\text{subst}' f) ee)$.

This completes Theorem 9 of the paper.

Lemma lam'_var (A B: Set)(f: A → B)(a: A) : lam' f (var' a) ≡ var' (f a).

Lemma lam'_app (A B: Set)(f: A → B)(t₁ t₂: Lam' A):

lam' f (app' t₁ t₂) ≡ app' (lam' f t₁)(lam' f t₂).

Lemma lam'_abs (A B: Set)(f: A → B)(r: Lam'(option A)):

lam' f (abs' r) ≡ abs' (lam' (option_map f) r).

Lemma lam'_flat (A B: Set)(f: A → B)(ee: Lam'(Lam' A)):

lam' f (flat' ee) ≡ flat' (lam' (lam' f) ee).

Lemma lam'_ext_refined (A B: Set)(f g: A → B)(t: Lam' A):

(∀ a, a occ' t → f a = g a) → lam' f t ≡ lam' g t.

Corollary lam'_ext (A B: Set)(f g: A → B):

(∀ a, f a = g a) → ∀ r, lam' f r ≡ lam' g r.

Lemma lam'_fct1 (A: Set)(t: Lam' A): lam' id t ≡ t.

Lemma lam'_fct2 (A B C: Set) (f: A → B) (g: B → C) (t: Lam' A):

lam' (g ∘ f) t ≡ lam' g (lam' f t).

Definition lift' (A B: Set)(f: A → Lam' B)(x: option A): Lam'(option B) :=

match x with

| None ⇒ var' None

| Some a ⇒ lam' Some (f a)

end.

Lemma lift'_ok (A B: Set)(f: A → Lam' B)(x: option A):

pi1 (lift' f x) = lift (pi1 ∘ f) x.

Lemma subst'Monad1 (A B: Set)(f: A → Lam' B)(a: A):

subst' f (var' a) ≡ f a.

Lemma subst'_var (A B: Set)(f: A → Lam' B)(a: A):

subst' f (var' a) ≡ f a.

Lemma subst'_app (A B: Set)(f: A → Lam' B)(t₁ t₂: Lam' A):

subst' f (app' t₁ t₂) ≡ app' (subst' f t₁)(subst' f t₂).

Lemma subst'_abs (A B: Set)(f: A → Lam' B)(r: Lam'(option A)):

subst' f (abs' r) ≡ abs' (subst' (lift' f) r).

Lemma subst'_flat (A B: Set)(f: A → Lam' B)(ee: Lam'(Lam' A)):

subst' f (flat' ee) ≡ flat' (subst' (var'(A := Lam' B) ∘ (subst' f)) ee).

Section 5.3 of the paper

Definition LamFmon2br: mon2br LamF.

Definition LamToCan: Lam ⊆ Lam := LamMMltDef.canonize LamFmon2br.

Lemma LamToCan_var (A: Set)(a: A): LamToCan(var a) = var a.

Lemma LamToCan_app (A: Set)(t₁ t₂: Lam A): LamToCan(app t₁ t₂) = app (LamToCan t₁) (LamToCan t₂).

Lemma LamToCan_abs (A: Set)(r: Lam (**option** A)): LamToCan(abs r) = abs(LamToCan r).

Lemma LamToCan_flat (A: Set)(ee: Lam (Lam A)):

LamToCan(flat ee) = flat (lam (LamToCan(A:= A)) (LamToCan ee)).

canonical elements are not changed by canonization

Theorem LamToCan_invariant (A: Set)(t: Lam A): **can** t → LamToCan t = t.

canonization yields hereditarily canonical elements

Theorem LamToCanCan (A: Set)(t: Lam A): **can** (LamToCan t).

canonization can be expressed as follows:

Definition LamToLam': Lam ⊆ Lam'.

Lemma LamToLam'_ok (A: Set)(t: Lam A): pi1 (LamToLam' t) = LamToCan t.

Scheme *canInd* := Induction for *can* Sort Prop.

Definition lsConstructed' (A: Set)(t: Lam' A) : Prop :=

(∃ a: A, t = **var**' a) ∨ (∃ t₁, ∃ t₂, t = **app**' t₁ t₂) ∨ (∃ r, t = **abs**' r) ∨ (∃ ee, t ≡ **flat**' ee).

Lemma Lam'exhausted (A: Set)(t: Lam' A): lsConstructed' t.

End LAMFLAT.