FORMALISING MONOTONE FRAMEWORKS

A dependently typed implementation in Agda

Master thesis

presented by Jorn van Wijk 3718778

Supervised by Jurriaan Hage and Wouter Swierstra

May 12, 2017

Utrecht University

» CONTENTS

AGDA

1999 *Agda*: Caterina Coquand

2007 *Agda 2*: Ulf Norell

Agda 2:

- Dependent type system
  - based on Martin Löf type theory
  - type can be dependent on values: `(x : A) → F x`
- Syntax similar to Haskell
- Functions are total

Curry Howard isomorhism

- type = proposition
- term = proof

Agda is used as proof assistant
Shares similarities with Coq, Epigram and NuPRL.
compilable to Haskell, Ocaml, javascript

Falsehood or non existence is represented by an uninhabitable type

```
data Empty where
```

Inductive datatype families are represented using a data block.

```
data Bool : Set where
 false : Bool
 true : Bool
```

We can use records to denote tuples with named fields (and several other features):

```
record Person : Set where
 field
  name : String
  age : ℕ
```

Implicit parameters (denoted by `{ }`) when inferable from context

```
id : {A : Set} → A → A
id {A} x = x

id false --> id {Bool} false
```

placeholders denoted by _ are used for fixity parsing.

e.g. `_≡_ : A → A → Set`

makes `≡` an infix operator.

Propositional equality is represented by an indexed datatype:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

```
unsound : false ≡ true
unsound = ?

sound : true ≡ true
sound = refl
```

First order logic quantifiers by Curry Howard correspondence:

Universal quantification (`∀`) can be encoded as a dependent function `∀ x ∈ A : P x --> (x : A) → P x`

We can use the dependent product (or record syntax) to represent existential quantification (`∃`): `∃ x ∈ A : P x`

```
record Σ (A : Set) (P : A → Set) : Set where
 field
  x : A
  Px : P x
```

These are defined in the Standard library.

ORDER THEORY

Partial ordered set

Set `ℂ` together with a binary relation `_⊑_ : ℂ → ℂ → Set`.
Requiring the following properties:

- Reflexivity = `{x : ℂ} → x ⊑ x`
- Transitivity = `{x y z : ℂ} → x ⊑ y → y ⊑ z → x ⊑ z`
- Antisymmetry = `{x y : ℂ} → x ⊑ y → y ⊑ x → x ≡ y`

From this, we can define a strict partial order as:

```
_⊏_ : ℂ → ℂ → Set
x ⊏ y = x ⊑ y ∧ x ≢ y
```

Poset has `⊤`, a.k.a. supremum / maximum. When:

```
(x : ℂ) → x ⊑ ⊤
```

`x` is upperbound of `S ⊆ ℂ`:

```
(s : S) → s ⊑ x
```

`y` is least upper bound (`⊔`) of `S ⊆ ℂ`:

```
(c : ℂ) → (s : S → s ⊑ c) → y ⊑ c
```

The least upper bound of `S`, if it exists, is referred to as `⊔ S`.

Join semi lattice poset `(ℂ,⊑)` with `⊔`

`⊔ : ℂ → ℂ → ℂ` is a binary total operator such that `x ⊔ y = ⨆ {x,y}`. It is bounded if it has a least element: `⊥` s.t.: `(c : ℂ) -> ⊥ ⊑ c`

Dually, we can define a lowerbound and a greatest lower bound (or meet; `⊓`) to form a (bounded) meet semi lattice.

Lattice poset that is a join semi lattice and meet semi lattice
Complete Lattice all subsets of `S ⊆ ℂ` have `⨆ S` defined.

When all possibly infinite sequences of form: $a_0 \sqsubseteq a_1 \sqsubseteq \cdots \sqsubseteq a_k \sqsubseteq \cdots$ eventually stabilize, i.e. : $\exists\ k \geq 0 : \forall\ j \geq k : a_j = a_k$ it is said that the poset satisfies the Ascending Chain Condition (ACC).

The ACC can be coded using well-foundedness. ACC on lattice implies it being complete.

Algebraic definition:

```
record BoundedSemiLattice a : Set (Level.suc a) where
  constructor boundedSemiLattice
  field
    ℂ : Set a -- Carrier type
    _⊔_ : ℂ → ℂ → ℂ -- Binary join
    _≟_ : (x y : ℂ) → Dec (x ≡ y) -- decidability of propositional equa
    ⊥ : ℂ -- Least element
    ⊥-isMinimal : (c : ℂ) -> ⊥ ⊑ c -- Proof that ⊥ is the least element
    ⊔-idem : (x : ℂ) → (x ⊔ x) ≡ x
    ⊔-comm : (x y : ℂ) → (x ⊔ y) ≡ (y ⊔ x)
    ⊔-cong₂ : {x y u v : ℂ} → x ≡ y → u ≡ v → (x ⊔ u) ≡ (y ⊔ v)
    ⊔-assoc : (x y z : ℂ) → ((x ⊔ y) ⊔ z) ≡ (x ⊔ (y ⊔ z))
    ⊐-isWellFounded : (x : ℂ) → Acc _⊐_ x
```

Additionally, from `⊔` and `≟` we define:

`⊑, ⊑?, ⊏, ⊏?, ⊒, ⊒?, ⊐ and ⊐?` .

```
-- properties about ⊔ and ⊑
⊔-on-⊑ : {a b c d : ℂ} → a ⊑ b → c ⊑ d → (a ⊔ c) ⊑ (b ⊔ d)
⊔-on-left-⊑ : {a b c : ℂ} → a ⊑ c → b ⊑ c  → a ⊔ b ⊑ c
⊔-on-right-⊑ : {a b c : ℂ} → a ⊑ b → a ⊑ b ⊔ c
left-⊔-on-⊑ : {a b : ℂ} → a ⊑ (a ⊔ b)
⊔-monotone-right : {x : ℂ} → Monotone _⊑_ (_⊔ x)
⊔-monotone-left : {x : ℂ} → Monotone _⊑_ (_⊔_ x)

-- properties about ⊑ and ≡
≡⇒⊑ : {a b : ℂ} → a ≡ b → a ⊑ b
⋢⇒≢ : {a b : ℂ} → ¬ (a ⊑ b) → ¬ a ≡ b
⊑-split-left :  {a b c : ℂ} → a ⊔ b ⊑ c → a ⊑ c
⊑-split-right :  {a b c : ℂ} → a ⊔ b ⊑ c → b ⊑ c

-- properties about ⊐ and ⊏
⊏-asymmetric, ⊏-trans, ⊐-trans, ...
```

TARSKI'S FIXED POINT THEOREM

1955: Alfred Tarski
formulated his fixed point theorem

He showed the existence of a fixed point (`f x ≡ x`) of any monotone function for any complete lattice.

```
Monotone : ∀{α ℓ} -> {ℂ : Set α} -> Rel ℂ ℓ ->
           (f : ℂ -> ℂ) -> Set (α Level.⊔ ℓ)
Monotone _⊑_ f = ∀ x y → x ⊑ y → f x ⊑ f y
```

Why is the least fixed point important?

to avoid superflouos information.

Given `f x = x ∪ {'a', 'b'}`,
`y = {'a', 'b', 'c'}` is a fixed point. But `{'a', 'b'}` is also a fixed point so `y` is not the least one.

```
IsFixedPoint : (c : ℂ) -> Set a
IsFixedPoint c = c ≡ f c

record FixedPoint : Set a where
 constructor fp
 field
  element : ℂ
  isFixedPoint : IsFixedPoint element
```

Our initial point `⊥` is extensive

```
fp-base : ⊥ ⊑ f ⊥
fp-base = ⊥-isMinimal (f ⊥)
```

Given an extensive point `c`, `f c` is also extensive

```
fp-step : ∀{c} -> c ⊑ f c → f c ⊑ f (f c)
fp-step = isMonotone
```

Given an extensive point, we find a fixed point by iteratively applying `f` .

```
l₀-isFixedPoint : {c : ℂ} -> c ⊑ f c -> FixedPoint
l₀-isFixedPoint {c} x with c ≟ f c -- are we there yet?
l₀-isFixedPoint {c} x | yes p = fp c p
l₀-isFixedPoint {c} x | no ¬p = l₀-isFixedPoint (fp-step x)

l₀ : FixedPoint
l₀ = l₀-isFixedPoint fp-base
```

Similarly:

```
IsLeastFixedPoint : (c : ℂ) -> Set a
IsLeastFixedPoint c = IsFixedPoint c
                    × ((e : FixedPoint) -> c ⊑ element e)

record LeastFixedPoint : Set a where
 constructor lfp
 field
  element : ℂ
  isLeastFixedPoint : IsLeastFixedPoint element
```

Suppose that `e` is a fixed point.

```
lfp-base : ⊥ ⊑ e
lfp-base = ⊥-isMinimal e
```

The inductive case:

```
lfp-step : {c : ℂ} -> c ⊑ e -> f c ⊑ e
lfp-step x = ⊑-trans (isMonotone x) (fixed⇒reductive p)
```

Finally, we obtain:

```
l₀-isLeastFixedPoint : {c : ℂ} -> c ⊑ f c
                     -> ((e : FixedPoint) -> c ⊑ element e )
                     -> LeastFixedPoint
l₀-isLeastFixedPoint {c} x x₁ with c ≟ f c
l₀-isLeastFixedPoint {c} x x₁ | yes p = lfp c (p , x₁)
l₀-isLeastFixedPoint {c} x x₁ | no ¬p =
  l₀-isLeastFixedPoint (fp-step x) (λ e → lfp-step e (x₁ e))
```

But unfortunately, we have no termination guarantee.

Agda's standard library offers us the accessibility predicate:

```
-- x is accessible if everything strictly
-- smaller than x is also accessible.
data Acc {a ℓ} {A : Set a} (_<_ : Rel A ℓ) (x : A) : Set (a ⊔ ℓ) where
  acc : (rs : ∀ y → y < x → Acc _<_ y) → Acc _<_ x

-- if all elements are accessible, then _<_ is well-founded.
Well-founded : ∀ {a ℓ} {A : Set a} → Rel A ℓ → Set _
Well-founded _<_ = ∀ x → Acc _<_ x
```

Our encoding of a bounded join semi lattice ensures well foundedness of `⊐`.

Use of the accessibility predicate:

```
l₀-isLeastFixedPoint : {c : ℂ} -> c ⊑ f c
                     -> ((e : FixedPoint) -> c ⊑ element e )
                     -> Acc _⊐_ c -> LeastFixedPoint
l₀-isLeastFixedPoint {c} p q g with c ≟ (f c)
l₀-isLeastFixedPoint {c} p q g | yes r = lfp c (r , q)
l₀-isLeastFixedPoint {c} p q (acc g) | no ¬r =
  l₀-isLeastFixedPoint (fp-step p)
                       (λ e → lfp-step e (q e))
                       (g (f c) (p , ¬r))
```

Select the next bigger element, accompanied by the proof that it is bigger.

Which we can invoke by starting from `⊥` and base cases:

```
l₀-lfp : LeastFixedPoint
l₀-lfp = l₀-isLeastFixedPoint fp-base lfp-base (⊐-isWellFounded ⊥)
```

LATTICE COMBINATORS

» LATTICE COMBINATORS

Manually creating the accessibility proofs for your structure can be challenging. Build up the proof using combinators. Each combinator is annotated with `ᴸ`.

- Unit: `Unitᴸ`
- Booleans: `Boolᴸ` = `Mayᴸ` , `Mustᴸ`
- Product (order): `_×ᴸ_`
- Biased sum: `⊎-leftᴸ` , `⊎-rightᴸ`
- N-ary product: `N-aryᴸ n L`
- Vector: `Vecᴸ L n`
- Powerset: `𝒫ᴸ L = 𝒫ᴸ-by-inclusion` , `𝒫ᴸ-by-exclusion`
- Total function space: `A -[ proof of A being finite ]→ L`

MONOTONE FRAMEWORKS

Monotone framework: Generalisation of types of source code analyses.

1. Assign labels to program points
2. *Control Flow Graph*
   - **Nodes**: Labels of program points
   - **Edges**: possible information flow during execution
3. pick initial point by label and assign initial value
4. define set of monotone transfer functions that take context and produce effect

Examples of analyses:

1. Live variables - what variables may be live (i.e. can be used in the future)
2. constant propagation - what variables have a constant value at this program point

Example of assigning labels:

```
fac : Stmt
fac = ”y” := var ”x”¹ seq
      ”z” := lit (+ 1)² seq
      while var ”y” gt lit (+ 1)³ do
      (
        ”x” := var ”z” mul var ”y”⁴ seq
        ”y” := var ”y” min lit (+ 1)⁵
      ) seq
      ”y” := lit (+ 0)⁶
```

Gives us control flow: `1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 3, 3 → 6`

We then define a monotone framework in Agda to be:

```
record MonotoneFramework a : Set (Level.suc a) where
 field
  n : ℕ -- number of labels
  L : BoundedSemiLattice a -- Lattice instance
 Label : Set
 Label = Fin n
 field
  𝓕 : Label -> ℂ -> ℂ  -- transfer functions indexed by label
  F : Graph n -- Control flow graph
  E : List Label -- Extremal labels
  ι : ℂ -- Extremal value
  𝓕-isMonotone : (ℓ : Fin n) → Monotone _⊑_ (𝓕 ℓ)
```

To compute the results we consider two values at every program point:

- Context:

```
analysis∘ ℓ′ = ⨆ { analysis• ℓ | ℓ ∈ predecessors F ℓ′ }
```

- Effect:

```
analysis• ℓ′ = 𝓕 ℓ′ (analysis∘ ℓ′)
```

We are looking for a fixed point in a vector structure x such that:

```
∀ ℓ → lookup ℓ x ≡ ⨆ { 𝓕 ℓ (analysis• ℓ) | ℓ ∈ predecessors F ℓ′ }
```

We can use the transfer functions `𝓕` and the flow `F` to compute a least fixed point. straight forward:

```
V× : BoundedSemiLattice _
V× = Vecᴸ L n ×ᴸ Vecᴸ L n

transfer-parallel : V×.ℂ → V×.ℂ
transfer-parallel (entry , exit) =
 let entry’ = 𝕍.map
      (λ ℓ′ → ιE ℓ′ ⊔ ⨆ (𝕃.map (flip lookup exit) (predecessors F ℓ′)))
      (allFin n)
 in (entry’ , (tabulate 𝓕 𝕍.⊛ entry’))

open TarskiFixedPointTheorem V× transfer-parallel transfer-parallel-isMo

parallel-lfp : LeastFixedPoint
parallel-lfp = l₀-lfp
```

Even more simple:

```
P : BoundedSemiLattice _
P = Label -[ _ ]→ L

parallel-tfs : P.ℂ → P.ℂ
parallel-tfs σ ℓ′ =
  ιE ℓ′ ⊔ ⨆ (𝕃.map (λ ℓ → ℱ ℓ (σ ℓ)) (predecessors F ℓ′))

open TarskiFixedPointTheorem P parallel-tfs parallel-tfs-isMonotone

parallel-tfs-lfp : LeastFixedPoint
parallel-tfs-lfp = l₀-lfp
```

Likewise, a more efficient algorithm (Chaotic iteration):

```
transfer-chaotic : Vec ℂ n × Vec ℂ n → Vec ℂ n × Vec ℂ n
transfer-chaotic x =
  𝕍.foldr (λ x₁ → _)
  (λ{ ℓ′ (entry , exit) →
     (let entry’ = ιE ℓ′ ⊔ ⨆ (𝕃.map (flip lookup exit) (predecessors F
      in (entry [ ℓ′ ]≔ entry’ , exit [ ℓ′ ]≔ ℱ ℓ′ entry’))})
  x
  (allFin n)

open TarskiFixedPointTheorem P transfer-chaotic transfer-chaotic-isMonot
```

The order is relevant.

```
W := nil;
for all (ℓ,ℓ′) ∈ F do
  W := cons((ℓ,ℓ′),W);
for all ℓ in F or E do
  if ℓ ∈ E then Analysis[ℓ] := ι
           else Analysis[ℓ] := ⊥;

while W ≠ nill do
  ℓ := fst(head(W));
  ℓ′  := snd(head(W));
  W := tail(W);
  if f[ℓ](Analysis[ℓ]) ⋢ Analysis[ℓ′] then
    Analysis[ℓ′] := Analysis[ℓ′] ⊔ f(Analysis[ℓ]);
    for all ℓ′′ with (ℓ′,ℓ′′) ∈ F do
      W := cons((ℓ′,ℓ′′),W);

for all ℓ ∈ F do
  MFP○(ℓ) := Analysis[ℓ];
  MFP●(ℓ) := f(Analysis[ℓ]);
```

The functional version of the worklist algorithm (MFP) in Agda:

```
mfp₁ : (x : ℂ) → (workList : List Edge) → ℂ
mfp₁ x [] = x
mfp₁ x ((l₁ , l₂) ∷ workList) with f l₁ x ⊑? lookup l₂ x
mfp₁ x ((l₁ , l₂) ∷ workList) | yes p = mfp₁ x workList
mfp₁ x ((l₁ , l₂) ∷ workList) | no ¬p =
 mfp₁ x' (lookup l₂ (adjacencyList F) 𝕃.++ workList)
  where x' : ℂ
        x' = x [ l₂ ]≔ f l₁ x ⊔ lookup l₂ x
```

Termination:

```
mfp₂ : (x : ℂ) → Acc _⊐_ x → (workList : List Edge) → ℂ
mfp₂ x x₁ [] = x
mfp₂ x x₁ ((l₁ , l₂) ∷ workList) with f l₁ x L.⊑? lookup l₂ x
mfp₂ x x₁ ((l₁ , l₂) ∷ workList) | yes p = mfp₂ x x₁ workList
mfp₂ x (acc rs) ((l₁ , l₂) ∷ workList) | no ¬p =
 mfp₂ x' (rs x' x⊏x') (lookup l₂ (adjacencyList F) 𝕃.++ workList)
  where x' : ℂ
        x' = x [ l₂ ]≔ f l₁ x ⊔ lookup l₂ x
        x⊏x' : x ⊏ x'
        x⊏x' = x⊑x' , x≢x'

result : ℂ
result = mfp₂ ⊥ (⊐-isWellFounded ⊥) F
```

```
worklist-theorem :
 -- for all vectors x
    (x : V.ℂ)
 -- that are below or equal to all other fixed points
  → (K : ((y : FixedPoint) → x V.⊑ fp y))
 -- and of which all greater values are accessible
  → Acc V._⊐_ x
 -- and is above or equal to the initial value
  → initial V.⊑ x
 -- and for all work lists
  → (workList : List Edge)
 -- that have all of their elements originating from the flow graph
  → ((e : Edge) → e ∈ workList → e ∈ F)
  → …
```

```
  -- and such that all two labels that form an edge in the flow graph are
   -- either contained in the work list or the value at ℓ′ in x is bigger
   -- the transfer function applied over the value at ℓ in x.
    → (I : ((ℓ ℓ′ : Label) → (ℓ , ℓ′) ∈ F →
       ((ℓ , ℓ′) ∈ workList) ⊎ (lookup ℓ′ x ⊒ 𝓕 ℓ (lookup ℓ x))))
   -- and such that for all ℓ′ the value at ℓ′ in x is less or equal to th
   -- of the transfer function applied over all predecessors and the init
   -- i.e. we stay below our definition of the fixed point
    → (J : ((ℓ′ : Label) →
             lookup ℓ′ x ⊑ lookup ℓ′ initial ⊔ ⨆ (𝕃.map (flip 𝑓 x) (predece
   -- there exists a fixed point, such that it is smaller than all other
    → Σ[ c ∈ FixedPoint ] ((y : FixedPoint) → fp c V.⊑ fp y)
```

Maintain invariants by lots of branching and rewriting:

```
begin
𝓕 ℓ (lookup ℓ (x [ ℓ′ ]≔ 𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x)) ⊔ lookup ℓ′ (x
≡⟨ cong (\i → 𝓕 ℓ (lookup ℓ (x [ ℓ′ ]≔ 𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x)) ⊔
𝓕 ℓ (lookup ℓ (x [ ℓ′ ]≔ 𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x)) ⊔ 𝓕 ℓ (lookup ℓ
≡⟨ cong (\i → 𝓕 ℓ i ⊔ 𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x) (lookup∘update′ ℓ≠
𝓕 ℓ (lookup ℓ x) ⊔ 𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x
≡⟨ sym (⊔-assoc _ _ _) ⟩
(𝓕 ℓ (lookup ℓ x) ⊔ 𝓕 ℓ (lookup ℓ x)) ⊔ lookup ℓ′ x
≡⟨ cong (\i → i ⊔ lookup ℓ′ x) (⊔-idem (𝓕 ℓ (lookup ℓ x))) ⟩
𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x
≡⟨ sym (lookup∘update ℓ′ x (𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x)) ⟩
lookup ℓ′ (x [ ℓ′ ]≔ 𝓕 ℓ (lookup ℓ x) ⊔ lookup ℓ′ x)
∎
```

Given a monotone framework, the algorithm results in a least fixed point:

```
lfp : Σ[ c ∈ FixedPoint ] ((y : FixedPoint) → fp c V.⊑ fp y)
lfp = worklist-theorem initial initial⊑fp (V.⊐-isWellFounded initial)
  V.⊑-reflexive F (λ e x → x) (λ ℓ ℓ′ x → inj₁ x) (λ ℓ′ → ⊔-on-right-⊑ ⊑
```

We use the While language as presented by Nielson, Nielson and Hankin[1].

The language consists of arithmetic and boolean expressions (simplified version):

```
data AExpr : Set where
  var : Ident → AExpr
  lit : ℤ → AExpr
  _plus_ : AExpr → AExpr → AExpr
  _min_ : AExpr → AExpr → AExpr
  _mul_ : AExpr → AExpr → AExpr
```

```
data BExpr : Set where
  true : BExpr
  false : BExpr
  not : BExpr → BExpr
  _and_ : BExpr → BExpr → BExpr
  _or_ : BExpr → BExpr → BExpr
  _gt_ : AExpr → AExpr → BExpr
```

```
data Stmt : Set where
  _:=_ : (v : String) → (e : AExpr) → Stmt
  skip :  Stmt
  _seq_ : (s₁ : Stmt) → (s₂ : Stmt) → Stmt
  if_then_else_ : (c : BExpr) → (t : Stmt) → (f : Stmt) → Stmt
  while_do_ : (c : BExpr) → (b : Stmt) → Stmt
```

We assign labels to statements to form program blocks:

```
data Stmt' : Set where
  _:=_ : (v : Var) → (e : AExpr) → (l : Lab) → Stmt'
  skip : (l : Lab) → Stmt'
  _seq_ : (s₁ : Stmt') → (s₂ : Stmt') → Stmt'
  if_then_else_ : (BExpr × Lab) → (t : Stmt') → (f : Stmt') → Stmt'
  while_do_ : (BExpr × Lab) → (b : Stmt') → Stmt'
```

Finally, we assume the program input for an analysis to be well-formed. `WhileProgram : Set` all labels are unique.

Furthermore, we define the following functions for a `WhileProgram`:

```
-- The initial label (entry point) of a statement
init : Stmt → Lab

-- The non empty set of final labels a statement can end
final : Stmt → List⁺ Lab

-- The set of labels for a statement
labels : Stmt → List Lab

-- The control flow graph, represented by a list of label pairs.
flow : Stmt → List (Lab × Lab)

-- Reversed flow
flowᴿ : Stmt → List (Lab × Lab)

-- variables of the program
Var* : Bag String
```

```
-- fv is a function that returns all free variables for some expression
fv : (BExpr | AExpr) → 𝓟 Var*

gen : Block → 𝓟 Var*
gen (skip l) = ⊥
gen ((x := a) l) = fv a
gen (bExpr c l) = fv c

kill : Block → 𝓟 Var*
kill (skip l) = ⊥
kill ((x := a) l) = ⁅ x ⁆
kill (bExpr c l) = ⊥
```

The transfer function can then be defined, for each label assuming Block is the block of the label, as:

```
transfer-function : Block → 𝒫 Var* → 𝒫 Var*
transfer-function b x = (x - (kill b)) ∪ gen b
```

Using these building blocks, we can form the monotone framework and perform the analysis:

```
live-variables : Stmt → MonotoneFramework _
live-variables program = record
  { L = 𝒫ᴸ Var*
  ; ℱ i = transfer-function (block i)
  ; F = flowᴿ program
  ; E = final program
  ; ι = ⊥
  ; ℱ-isMonotone = postulate
  }
```

Note that live variable analysis is a backward analysis, which we perform by using the reversed flow: `flowᴿ` and by starting from the final labels.

Available Expression Analysis computes at every program point what subexpressions are available. It is also a kill-gen analysis.

```
available-expressions : MonotoneFramework _
available-expressions = record
 { L = 𝒫ᴸ-by-exclusion (length AExpr⋆)
 ; ℱ = transfer-functions
 ; F = flow labelledProgram
 ; E = [ init labelledProgram ]
 ; ι = ⊥
 ; ℱ-isMonotone = postulate
 }
```

Note that we now make use of the normal flow and start at the initial label of our program.

Constant propagation, a forward analysis using the total function space: For each variable: what value can it be?

```
constant-propagation : Stmt → MonotoneFramework _
constant-propagation program = record
 { L = Fin m -[ m , Inverse.id ]→ ℤ⊤⊥ᴸ
 ; 𝓕 = transfer-functions
 ; F = flow labelledProgram
 ; E = Data.List.[ init program ]
 ; ι = λ x → top
 ; 𝓕-isMonotone = postulate
 }
```

EMBELLISHED MONOTONE FRAME-WORKS

So far, only simple non procedural language. Luckily, we do not have to modify the `mfp` algorithm

instead: represent embellished as regular framework.

```
asMonotoneFramework : EmbellishedMonotoneFramework _ → MonotoneFramework
```

Update statement

```
data Stmt : Set where
  call : (name : String) → (arguments : List AExpr) → (result : String)
  ...
```

We assign two labels to call: call and return.

Add declarations:

```
data Decl : Set where
  proc_⟨_,_⟩_end : (name : String) → (arguments : List String) →
                  (result : String) → (body : Stmt) → Decl
data Program : Set where
  begin_main-is_end : (declarations : List Decl) → (main : Stmt) → Prog
```

We call the resulting language While-Fun.

Also:

```
init⋆  : Program → Lab
final⋆ : Program → List Lab
flow⋆  : Program → List Edge
```

Validity constraints:

- referenced procedure calls must be defined
- all procedures must be uniquely named

Like Nielson, Nielson and Hankins: Use abstract call stacks to make sure information flows through valid paths:

```
Δ : Set
Δ = BoundedList Label k
```

A call string represents the top of a call stack (of length `k`) at each program point.

At call and return labels we select only valid paths.

To model the call stack we use the total function space (so the call string must be finite):

```
L̂ : BoundedSemiLattice a
L̂ = Δ -[ .. ]→ L
```

Information at the return label of a call:

- local scope
- from the call

use Agda's dependent type system:

```
𝓕 : (l : Label) -> arityToType (arity labelType l) (BoundedSemiLattice.
```

Also adjust monotonicity requirement:

```
BiMonotone = ∀{x y z w} -> x ⊑ y → z ⊑ w -> f x z ⊑ f y w
```

We then form an EmbellishedMonotoneFramework:

```
constant-propagation-embellished : EmbellishedMonotoneFramework _
constant-propagation-embellished = record
  { n = n
  ; L = L
  ; k = 2
  ; labelType = embellishedType
  ; 𝓕 = transfer-emb
  ; F = flow⋆ labelledProgram
  ; E = Data.List.[ init⋆ labelledProgram ]
  ; ι = λ x → top
  ; 𝓕-isMonotone = postulate
  }
```

Note: `𝓕` acts on `L` (instead of `L̂`), so proofs are a lot easier.

EXTENDED MONOTONE FRAMEWORKS

Dynamicly typed languages such as Python. Control Flow Graph unknown staticly.

Compute CFG using Lattice during `mfp` algorithm.

```
  Fᴸ : BoundedSemiLattice _
    Fᴸ = 𝒫ᴸ-by-inclusion (n * n)
```

use a `next` function, that supplies new edges.

Simple version without proof:

```
  mfp-extended :
   -- for all vectors x
      (x : V.ℂ)
   -- and for all work lists
    → (workList : List (Label × Label))
   -- and for all control flow graphs
    → (F : CFG)
   -- there exists a control flow graph F̂ such that we obtain an x.
    → Σ[ F̂ ∈ CFG ] V.ℂ
```

For regular monotone frameworks, as well as embellished ones.

» RELATED WORK

- *David Darais and David Van Horn*
  Abstract interpreter correctness by construction monadically composed galois connections.
- *J. Knoop et al* machine checkable abstract interpretation based interprocedural data flow analysis in the theorem prover Athena
- *David Cachera et al*
  provide a similar framework in Coq with a constraint based analysis for OCaml.

- *Kahl and Al-hassy* Relational Algebraic Theories in Agda (RATH)
  Dualisation techniques: > 52GB of heapspace (on a machine with 64GB of RAM) and at least a few days to type check.
- *Compcert project* possibilities of compiler verification
  Compcert C compiler: 90% of the algorithms are verified (Coq).
  Kildall's algorithm for
  - constant propagation
  - dead code elimination
  - common subexpression elimination

» CONCLUSION

Formalized:

- Tarkski's theorem
- Monotone frameworks
- Parallel and chaotic iteration and MFP
- Embellished frameworks
- Extended frameworks
- Set of combinators to ensure termination

FURTHER RESEARCH

- Dependently typed attribute grammar
- Real life examples
- model MFP as monotone function

» THANKS FOR YOUR ATTENTION

Agda code, thesis and references:

`github.com/jornvanwijk/monotoneframeworks-agda`

presentation theme based on mtheme by Matthias Vogelgesang

Contact me: jornvanwijk@gmail.com / J.J.vanWijk@students.uu.nl