Verified Graph Rewriting

Dependent types in general programming

Giuseppe Lomurno

4 October 2019

Università di Pisa

Table of contents

Motivation

- Exploration of dependent types in general programming
- Respect the formal specification
- Express complex invariants
- Reduce runtime errors
- Avoid duplication in coding and validation efforts

- Exploration of dependent types in general programming
- Respect the formal specification
- Express complex invariants
- Reduce runtime errors
- Avoid duplication in coding and validation efforts

- Exploration of dependent types in general programming
- Respect the formal specification
- Express complex invariants
- Reduce runtime errors
- Avoid duplication in coding and validation efforts

- Exploration of dependent types in general programming
- Respect the formal specification
- Express complex invariants
- Reduce runtime errors
- Avoid duplication in coding and validation efforts

- Exploration of dependent types in general programming
- Respect the formal specification
- Express complex invariants
- Reduce runtime errors
- Avoid duplication in coding and validation efforts

Alternatives

- Empirical and property testing
- Formal methods
- ADTs and polymorphism

Alternatives

- Empirical and property testing
- Formal methods
- ADTs and polymorphism

Alternatives

- Empirical and property testing
- Formal methods
- ADTs and polymorphism

DEPENDENT TYPES

Dependent Types Theory

Judgements

Basic kind of judgements in dependent theories:

$\vdash \Gamma$ context	Γ is a well-formed context
$\Gamma \vdash \sigma$ type	σ is a type in Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in Γ
$\vdash \Gamma = \Delta$ context	Γ and Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ and τ are definitionally equal types
$\Gamma \vdash M = N : \sigma$	M, N are definitionally equal terms

Dependent function space

Also known as Π-types, they are function with return type dependent on the value of its arguments

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Pi x \colon \sigma.\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type} \qquad \Gamma, x : \sigma \vdash M : \tau[x]}{\Gamma \vdash \lambda \colon \sigma.\, M : \Pi x \colon \sigma.\, \tau[x]}$$

$$\frac{\Gamma \vdash \lambda x \colon \sigma.\, M : \Pi x \colon \sigma.\, \tau[x] \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{App}(\lambda x \colon \sigma.\, M, N) = M[x \coloneqq N] : \tau[N]}$$

Also known as Π-types, they are function with return type dependent on the value of its arguments

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Pi x \colon \sigma .\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type} \qquad \Gamma, x : \sigma \vdash M : \tau[x]}{\Gamma \vdash \lambda \colon \sigma .\, M : \Pi x \colon \sigma .\, \tau[x]}$$

$$\frac{\Gamma \vdash \lambda x \colon \sigma .\, M : \Pi x \colon \sigma .\, \tau[x] \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{App}(\lambda x \colon \sigma .\, M, N) = M[x \coloneqq N] : \tau[N]}$$

Dependent function space

Also known as Π-types, they are function with return type dependent on the value of its arguments

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Pi x \colon \sigma.\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type} \qquad \Gamma, x : \sigma \vdash M : \tau[x]}{\Gamma \vdash \lambda \colon \sigma.\, M : \Pi x \colon \sigma.\, \tau[x]}$$

$$\frac{\Gamma \vdash \lambda x \colon \sigma.\, M : \Pi x \colon \sigma.\, \tau[x] \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{App}(\lambda x \colon \sigma.\, M, N) = M[x \coloneqq N] : \tau[N]}$$

Dependent sum

Also known as Σ-types, represent a pair where the second projection depends on the first

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Sigma x \colon \sigma.\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau[M]}{\Gamma \vdash \langle M, N \rangle : \Sigma x \colon \sigma.\, \tau[x]}$$

$$\frac{\begin{array}{l}\Gamma, z : \Sigma x \colon \sigma.\, \tau \vdash \rho[z] \text{ type} \\ \Gamma, x : \sigma, y : \tau \vdash H : \rho[\langle x, y \rangle] \qquad \Gamma \vdash M : \Sigma x \colon \sigma.\, \tau[x]\end{array}}{\Gamma \vdash \mathrm{R}^{\Sigma}(H, M) : \rho[M]}$$

Dependent sum

Also known as Σ-types, represent a pair where the second projection depends on the first

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Sigma x \colon \sigma.\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau[M]}{\Gamma \vdash \langle M, N \rangle : \Sigma x \colon \sigma.\, \tau[x]}$$

$$\frac{\begin{array}{c}\Gamma, z : \Sigma x \colon \sigma.\, \tau \vdash \rho[z] \text{ type} \\ \Gamma, x : \sigma, y : \tau \vdash H : \rho[\langle x, y \rangle] \qquad \Gamma \vdash M : \Sigma x \colon \sigma.\, \tau[x]\end{array}}{\Gamma \vdash R^{\Sigma}(H, M) : \rho[M]}$$

Dependent sum

Also known as Σ-types, represent a pair where the second projection depends on the first

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Sigma x \colon \sigma.\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau[M]}{\Gamma \vdash \langle M, N \rangle : \Sigma x \colon \sigma.\, \tau[x]}$$

$$\frac{\begin{array}{l}\Gamma, z : \Sigma x \colon \sigma.\, \tau \vdash \rho[z] \text{ type} \\ \Gamma, x : \sigma, y : \tau \vdash H : \rho[\langle x, y \rangle] \qquad \Gamma \vdash M : \Sigma x \colon \sigma.\, \tau[x]\end{array}}{\Gamma \vdash \mathrm{R}^{\Sigma}(H, M) : \rho[M]}$$

Propositional equality

Definitional equality is only in form of judgements, we can define propositional equality

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{Id}_\sigma(M, N) \ \mathsf{type}}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{Refl}_\sigma(M) : \mathsf{Id}_\sigma(M, M)}$$

Definitional equality is only in form of judgements, we can define propositional equality

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{Id}_\sigma(M, N)\ \text{type}}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{Refl}_\sigma(M) : \mathrm{Id}_\sigma(M, M)}$$

Curry-Howard correspondence

First order logic	Dependent type model
Proposition	σ
Proof	$M : \sigma$
Predicate	$\tau[x]$
$\forall x \in \sigma.\, \tau$	$\Pi x\colon \sigma.\, \tau[x]$
$\exists x \in \sigma.\, \tau$	$\Sigma x\colon \sigma.\, \tau[x]$
$\sigma \Rightarrow \tau$	$\sigma \to \tau$
$\sigma \wedge \tau$	$\sigma \times \tau$
$\sigma \vee \tau$	$\sigma + \tau$
$\neg\sigma$	$\sigma \to \bot$
True	$\top$
False	$\bot$
$M = N$	$\mathrm{Id}_\sigma(M, N)$

Caveat: Functions must be total

Idris

Haskell-like syntax with GADTs definitions

```
data Nat = S | Z Nat
data Fin : (n : Nat) -> Type where
  FZ : Fin Z
  FS : Fin n -> Fin (S n)
```

Native support to dependent functions

```
size : VSub xs -> Nat
size Empty = 0
size (In sub) = 1 + size sub
size (Out sub) = size sub

extract : (xs : Vect n a) -> (sub : VSub xs)
       -> Vect (size sub) a
extract [] Empty = []
extract (x :: xs) (In sub) = x :: extract xs sub
extract (x :: xs) (Out sub) = extract xs sub
```

Dependent pairs for sigma types

```
filter : (elem -> Bool)
      -> Vect len elem
      -> (n : Nat ** Vect n elem)
filter p [] = (0 ** [])
filter p (x :: xs) = case filter p xs of
  (n ** tail) => if p x
                    then (S n ** x :: tail)
                    else (n ** tail)
```

Type for equality and equality rewritings

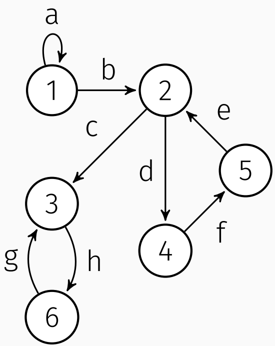
```
data (=) : (x : A) -> (y : B) -> Type where
  Refl : x = x

lemma : (n, m : Nat) -> S (n + m) = n + (S m)
lemma Z m = Refl
lemma (S n) m = let hyp = lemma n m in
                    rewrite hyp in Refl
```

A case study

Directed graphs

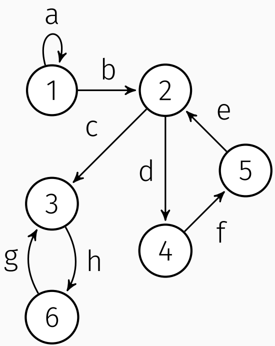
Rewriting directed graphs using a categorical approach



- Simple to understand
- Nice algebraic representation
- Almost direct translation
- Non trivial properties
- Interest in the field

Directed graphs

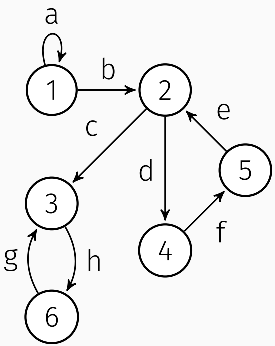
Rewriting directed graphs using a categorical approach



- Simple to understand
- Nice algebraic representation
- Almost direct translation
- Non trivial properties
- Interest in the field

Directed graphs

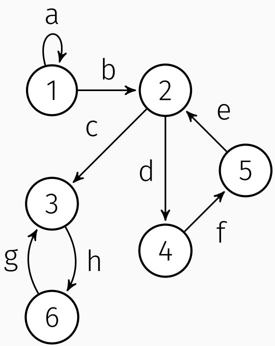
Rewriting directed graphs using a categorical approach



- Simple to understand
- Nice algebraic representation
- Almost direct translation
- Non trivial properties
- Interest in the field

Directed graphs

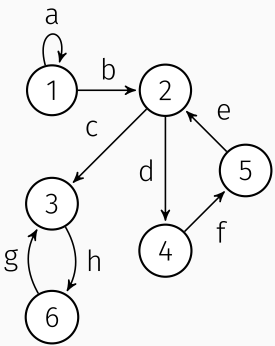
Rewriting directed graphs using a categorical approach



- Simple to understand
- Nice algebraic representation
- Almost direct translation
- Non trivial properties
- Interest in the field

Directed graphs

Rewriting directed graphs using a categorical approach



- Simple to understand
- Nice algebraic representation
- Almost direct translation
- Non trivial properties
- Interest in the field

Inductive definitions

```
data Graph : (n, m : Nat) -> {vertex, edge : Type}
          -> (vs : Vect n vertex) -> (es : Vect m edge)
          -> Type where
  Empty : Graph n Z vs []
  Edge  : (s, t : Fin n) -> Graph n m vs es
       -> Graph n (S m) vs (e :: es)
```

- Follows natural deduction reasoning
- Richness in property expressivity
- Easily translatable to common representations
- Performance analysis sidelined at the moment!

Inductive definitions

```
data Graph : (n, m : Nat) -> {vertex, edge : Type}
          -> (vs : Vect n vertex) -> (es : Vect m edge)
          -> Type where
  Empty : Graph n Z vs []
  Edge  : (s, t : Fin n) -> Graph n m vs es
       -> Graph n (S m) vs (e :: es)
```

- Follows natural deduction reasoning
- Richness in property expressivity
- Easily translatable to common representations
- Performance analysis sidelined at the moment!

Inductive definitions

```
data Graph : (n, m : Nat) -> {vertex, edge : Type}
          -> (vs : Vect n vertex) -> (es : Vect m edge)
          -> Type where
  Empty : Graph n Z vs []
  Edge  : (s, t : Fin n) -> Graph n m vs es
       -> Graph n (S m) vs (e :: es)
```

- Follows natural deduction reasoning
- Richness in property expressivity
- Easily translatable to common representations
- Performance analysis sidelined at the moment!

Inductive definitions

```
data Graph : (n, m : Nat) -> {vertex, edge : Type}
          -> (vs : Vect n vertex) -> (es : Vect m edge)
          -> Type where
  Empty : Graph n Z vs []
  Edge  : (s, t : Fin n) -> Graph n m vs es
       -> Graph n (S m) vs (e :: es)
```

- Follows natural deduction reasoning
- Richness in property expressivity
- Easily translatable to common representations
- Performance analysis sidelined at the moment!

Expressing properties

Properties of the graph modeled both as dependent functions

```
indegree : (v : Fin n) -> (g : Graph n m vs es) -> Nat
predecessors : (v : Fin n) -> (g : Graph n m vs es)
            -> Vect (indegree v g) (Fin n)
```

and predicates

```
data HasEdge : (s, t : Fin n) -> (g : Graph n m vs es)
            -> Type where
  Here  : HasEdge s t (Edge s t g)
  There : HasEdge s t g -> HasEdge s t (Edge s' t' g)

edgeIndex : (s, t : Fin n) -> (g : Graph n m vs es)
         -> {auto prf : HasEdge s t g} -> Fin m
edgeIndex s t (Edge s t g) {prf = Here} = FZ
edgeIndex s t (Edge _ _ g) {prf = There prf}
  = FS (edgeIndex s t g)
```

Morphisms

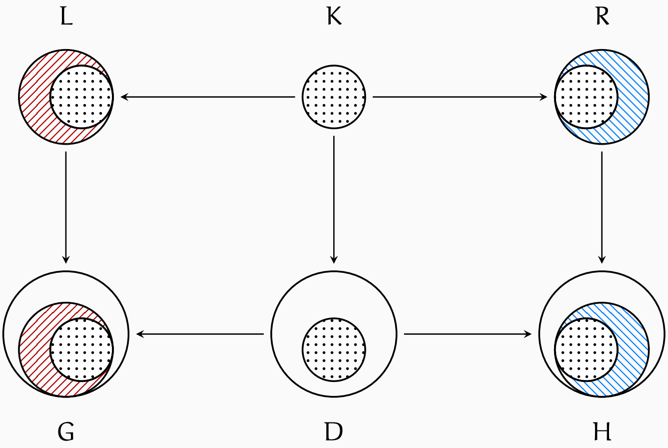
Assuming finite vertices and edges sets we represent morphisms functions (f_E, f_V) as vectors

$$\begin{tikzcd} E_1 \arrow[r, shift left, "s_1"] \arrow[r, shift right, "t_1"'] \arrow[d, "f_E"'] & V_1 \arrow[d, "f_V"] \\ E_2 \arrow[r, shift left, "s_2"] \arrow[r, shift right, "t_2"'] & V_2 \end{tikzcd}$$

```
data Morphism : (g  : Graph n  m  vs  es)
             -> (g' : Graph n' m' vs' es')
             -> Type where
  Morph : {g : Graph n m vs es}
       -> {g' : Graph n' m' vs' es'}
       -> (vmap : Vect n (Fin n'))
       -> (emap : Vect m (Fin m'))
       -> Morphism g g'

data PreserveSource : (g  : Graph n  m  vs  es)
                   -> (g' : Graph n' m' vs' es')
                   -> (mor : Morphism g g') -> Type
                        where
  Empty : PreserveSource Empty g' mor
  Edge  : source e g' = index s vmap -> -- ...
```

Simple rewrite rule



Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

Rewrite phases

1. Injectivity and rule check
2. Morphisms check
3. Subgraph matching
4. Subgraph extraction
5. Rule merging
6. Additional morphisms check
7. Commutativity checks

DEMO

Conclusions

Pros

No duplication of code and verification model

Types as specification

Less time in testing phase

Type refinement

Restrictive specifications lead to ”automatic” coding

Cons

Theorem proving is not simple

Cryptic error messages

Tooling and ecosystem

Pitfalls

Reduced inference

Isomorphic representation have non similar difficulties

Totality

Performance and runtime erasure

Medio tutissimus ibis

Views on existing representations

Incremental development with deferred code

Opt-in totality and laziness

Compatible with existing regular functional code