Containers

based on joint work with Michael Abbott, Neil Ghani, Peter Hancock, Conor McBride, Peter Morris

Thorsten Altenkirch

School of Computer Science and IT
University of Nottingham

May 16, 2008

Overcoming the ASCII-greek dichotomy

- Containers: Theory of (concrete) datatypes
- See Michael Abbott's PhD and our joint papers
  e.g. *Containers - Constructing Strictly Positive Types* (TCS 2005)
- Using Category Theory and Type Theory
- Application: Generic Programming and Theorems
- How to explain it to (functional) programmers?
- Today: Use Agda to implement Containers

Why Containers ?

- Conceptually: What is a datatype?
- Categorically: A category with all inductive types?
- Generic programs
- Generic theorems
- Small trusted cores
- Reasoning about polymorphic functions

Agda ?

- Dependently typed programming language
- Implemented by Ulf Norell (Chalmers)
- Inspired by Cayenne and Epigram

What is a datatype?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data [_] (a : Set) : Set where
  []  : [ a ]
  _::_ : a → [ a ] → [ a ]

data T : Set where
  node : [ T ] → T

data Tω : Set where
  leaf : Tω
  nodeω : (ℕ → Tω) → Tω
```

Dubious candidates (not accepted by Agda)

- Negative types

```
data D : Set where
  fun : (D -> D) -> D
```

- Non-strict positive types

```
data R : Set where
  r : ((R -> ℕ) -> ℕ) -> R
```

Inductive families

```
data Fin : ℕ -> Set where
  fz : forall {n} -> Fin (suc n)
  fs : forall {n} -> Fin n -> Fin (suc n)

fmax : { n : ℕ } -> Fin (suc n)
fmax {zero} = fz
fmax {suc n} = fs (fmax {n})

femb : { n : ℕ } -> Fin n -> Fin (suc n)
femb fz = fz
femb (fs i) = fs (femb i)

finv : { n : ℕ } -> Fin n -> Fin n
finv fz = fmax
finv (fs i) = femb (finv i)
```

(Unary) container

- A container is given by:
  - a set of shapes
  - a family of positions

```
data Cont : Set where
  _◁_ : (S : Set) -> (S -> Set) -> Cont

CList : Cont
CList = ℕ ◁ Fin
```

Extension of a container

- Every container gives rise to a functor.
- A function on types:

```
data ⟦_⟧ : Cont -> Set -> Set where
  _◁₀_ : {S : Set} {P : S -> Set}{ X : Set }
         -> (s : S)
         -> ((P s) -> X)
         -> ⟦ (S ◁ P) ⟧ X
```

- a map function:

```
⟦_⟧₀ : (SP : Cont) -> {X Y : Set}
       -> (X -> Y)
       -> ⟦ SP ⟧ X -> ⟦ SP ⟧ Y
⟦ S ◁ P ⟧₀ f (s ◁₀ g) = s ◁₀ \ p -> f (g p)
```

Reconstructing lists

```
nil : {A : Set} -> ⟦ CList ⟧ A
nil {A} = 0 ◁₀ as
          where as : Fin 0 -> A
                as ()

cons : {A : Set} -> A ->  ⟦ CList ⟧ A ->  ⟦ CList ⟧ A
cons {A} a (n ◁₀ as) = (suc n) ◁₀ aas
                       where aas : Fin (suc n) -> A
                             aas fz = a
                             aas (fs i) = as i
```

Derivatives

- Example of a generic operation.
- The derivative of a parametric type is the type *of its one-hole context*.
- Important ingredient of the generic zipper.

```
data _≡_ {A : Set} : A -> A -> Set where
  refl : {a : A} -> a ≡ a

_!≡_ : {A : Set} -> A -> A -> Set
a !≡ b = (a ≡ b) -> ⊥

D : Cont -> Cont
D (S ◁ P) = Σ S P ◁ \ sp -> Σ (P (proj₁ sp)) (\ p -> p !≡ proj₂ sp)
```

Operations on container

- Containers are closed under:
  - disjoint union (coproducts, Either in Haskell)
  - products
  - constant function types

```
_⊎₀_ : Cont → Cont → Cont
(S ◁ P) ⊎₀ (T ◁ Q) = (S ⊎ T) ◁ [ P , Q ]

_×₀_ :  Cont → Cont → Cont
(S ◁ P) ×₀ (T ◁ Q) = (S × T) ◁ (\ st → P (proj₁ st) ⊎ Q (proj₂ st))

_⇒_ : Set → Cont → Cont
X ⇒ (S ◁ P) = (X → S) ◁ \ f → Σ X (\ x → P (f x))
```

Dependent product

- Dependent products generalize prodducts and constant function spaces.

```
_^S : Cont -> Set
_^S (S ◁ P) = S

_^P : (SP : Cont) -> SP ^S -> Set
_^P (S ◁ P) = P

Π₀ : (X : Set) -> (X -> Cont) -> Cont
Π₀ X F = ((x : X) -> (F x)^S) ◁ \ f -> Σ X (\ x -> ((F x)^P) (f x))

_×₁_ :  Cont -> Cont -> Cont
SP ×₁ TQ = Π₀ Bool (\ b -> if b then SP else TQ)

_⇒₀_ : Set -> Cont -> Cont
X ⇒₀ TQ = Π₀ X (\ x -> TQ)
```

Fixpoints

- Fixpoints of containers always exist.
- Categorically: initial algebras

```
data W (S : Set) (P : S -> Set) : Set where
  sup : (s : S) -> ((P s) -> W S P) -> W S P

μ : Cont -> Set
μ (S ◁ P) = W S P

T₀ : Set
T₀ = μ CList

node₀ : ⟦ CList ⟧ T₀ -> T₀
node₀ (n ◁₀ g) = sup n g
```

Morphisms of containers

- Morphisms model polymorphic functions (*natural transformations*)
- A morphism is given by:
  - a function on shapes
  - a backward function on positions
- Theorem: every polymorphic function is given by a container morphism.

```
data _⟹_ : Cont → Cont → Set where
  _◁₁_ : forall {S P T Q} → (f : S → T)
                          → ({s : S} → Q (f s) → P s)
                          → (S ◁ P) ⟹ (T ◁ Q)

⟦_⟧₁ : forall {SP TQ} → (SP ⟹ TQ)
                      → {X : Set} → (⟦ SP ⟧ X) → ⟦ TQ ⟧ X
⟦ f ◁₁ r ⟧₁ {X} (t ◁₀ h) = (f t) ◁₀ (\ q →  h (r {t} q)  )
```

n-ary containers

- *n*-ary containers model *n*-ary functors.

```
data Cont (n : ℕ) : Set where
  _◁_ : (S : Set)
        -> ((Fin n) -> S -> Set)
        -> Cont n

data ⟦_⟧ {n : ℕ} : Cont n -> (Fin n -> Set) -> Set where
  _◁₀_ : {S : Set} {Ps : Fin n -> S -> Set}{ Xs : Fin n -> Set }
         -> (s : S)
         -> ({i : Fin n} -> Ps i s -> Xs i)
         -> ⟦ (S ◁ Ps) ⟧ Xs
```

Parametrized fixpoints

- *n*-ary containers are closed under fixpoints.

```
data W (S : Set) (P : S -> Set) : Set where
  sup : (s : S) -> ((P s) -> W S P) -> W S P

data Pos {n : ℕ}(S : Set)(Ps : Fin (suc n) -> S -> Set)
         (i : Fin n) : W S (Ps fz) -> Set where
  here : {s : S}{f : Ps fz s ->  W S (Ps fz)}
         -> (Ps (fs i) s)
         -> Pos S Ps i (sup s f)
  below : {s : S}{f : Ps fz s ->  W S (Ps fz)}
          -> (q : Ps fz s) -> (Pos S Ps i (f q))
          -> Pos S Ps i (sup s f)

μ : {n : ℕ} -> Cont (suc n) -> Cont n
μ (S ◁ Ps) =   W S (Ps fz)  ◁  Pos S Ps
```

Indexed container

- No need to limit to finite index sets.
- Indexed containers can model inductive families (like Fin).
- Dependent types can *eat* themselves.

```
data Cont (I : Set) : Set where
  _◁_ : (S : Set)
        -> (I -> S -> Set)
        -> Cont I

data ⟦_⟧ {I : Set} : Cont I -> (I -> Set) -> Set where
  _◁₀_ : {S : Set} {Ps : I -> S -> Set}{ Xs : I -> Set }
         -> (s : S)
         -> ({i : I} -> Ps i s -> Xs i)
         -> ⟦ (S ◁ Ps) ⟧ Xs
```

Loose ends

- Coinductive types
- Inductive definitions of type universes:
  - Strictly positive types
  - Strictly positive families
- Hierarchy of container universes:
  - Small containers (finite position sets)
  - Differentiable container (decidable equality on position)
- See our lecture notes on
  Generic Programming with Dependent Types
  using Epigram instead of Agda.