Correctness of Object Oriented Models by Extended Type Inference

Simon Foster[1], Georg Struth[2] and Ondrej Rypáček[2]

University of York[1]
University of Sheffield[2]

September 25, 2012

Motivation

- class diagrams are important structures in software engineering
- used extensively in the MDA to define data requirements
- logical constraints on data define correctness (e.g. OCL)
- how to ensure models are correct?
- how to ensure model transformations preserve constraints?
- our aim: MDD in dependent type theory
- contribution: object graphs correct by construction

Agda

- a dependently typed functional programming language
- like Haskell with dependent types
- contains a proof-assistant for discharging complex constraints
- based on Martin Löf type theory
- developed at Chalmers University by Norrell et al.
- based on a previous ITP also called Agda
  (technically, this is Agda 2)
- like Coq but more programming oriented
- type-checker doubles as a proof checker
- can be used as a specification language

Our approach

- we give an experimental encoding of class and object diagrams
- our code has been simplified for the paper and presentation
- class and objects diagrams formalised using dependent records
- records act as templates which must be type-correct
- these templates can be filled in gradually in Agda
- this supports incremental development
- each object graph must respect its class graph
- type theory binds the two together
- parts of the structure can be type inferred

Dependent Records

- an implementation of $\Pi\Sigma$ types
- a declaration consists of
  - a type signature, which may be parametric (Π)
  - a collection of ordered fields (Σ)
- the type of each field can depend on prior fields
- allows types with propositions (correct by construction)

Dependent Records

```
record Person : Set where
  field
    name     : String
    age      : ℕ
    ageValid : age ⩽ 200

record Employee : Set where
  field
    person : Person
  open Person person
  field
    yearsWorking : ℕ
    yearsValid   : (age - yearsWorking) ⩾ 16
```

Class Graphs

$$
\begin{array}{l}
\textbf{record}\ \mathsf{ClassGraph}\ (\mathsf{Types}\ :\ \mathsf{Set})\ :\ \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\qquad \begin{array}{ll}
\mathsf{Class} & : \mathsf{Set} \\
\mathsf{Attr} & : \mathsf{Class} \to \mathsf{Set} \\
\mathsf{Assoc} & : \mathsf{Class} \to \mathsf{Set} \\
\mathsf{attrType} & : \{\mathsf{c}\ :\ \mathsf{Class}\} \to \mathsf{Attr}\ \mathsf{c} \to \mathsf{Types} \\
\mathsf{assocRange} & : \{\mathsf{c}\ :\ \mathsf{Class}\} \to \mathsf{Assoc}\ \mathsf{c} \to \mathsf{Interval} \\
\Delta & : \{\mathsf{c}\ :\ \mathsf{Class}\} \to \mathsf{Assoc}\ \mathsf{c} \to \mathsf{Class}
\end{array}
\end{array}
$$

$$
\mathsf{IsInRange}\ :\ \mathsf{Interval} \to \mathbb{N} \to \mathsf{Set}
$$

Object Graphs

```
record ObjGraph {T} (G : ClassGraph T) (⟦_⟧ : T → Set) : Set₁ where
  open ClassGraph G
  field
    Obj      : Class → Set
    attrVals : ∀ {c} (o : Obj c) (a : Attr c) → ⟦ attrType a ⟧
    δ        : ∀ {c} (o : Obj c) (a : Assoc c)
               → RVec (Obj (Δ a)) (assocRange a)
```

Object Graphs

- Objects

```
record ObjGraph {T} (G : ClassGraph T) (⟦_⟧ : T → Set) : Set₁ where
  open ClassGraph G
  field
    Obj     : Class → Set
    attrVals : ∀ {c} (o : Obj c) (a : Attr c) → ⟦ attrType a ⟧
    δ       : ∀ {c} (o : Obj c) (a : Assoc c)
              → RVec (Obj (Δ a)) (assocRange a)
```

Object Graphs

- Objects
- Attributes

```
record ObjGraph {T} (G : ClassGraph T) (⟦_⟧ : T → Set) : Set₁ where
  open ClassGraph G
  field
    Obj     : Class → Set
    attrVals : ∀ {c} (o : Obj c) (a : Attr c) → ⟦ attrType a ⟧
    δ       : ∀ {c} (o : Obj c) (a : Assoc c)
              → RVec (Obj (Δ a)) (assocRange a)
```

Object Graphs

- Objects
- Attributes
- Associations

```
record ObjGraph {T} (G : ClassGraph T) (⟦_⟧ : T → Set) : Set₁ where
  open ClassGraph G
  field
    Obj      : Class → Set
    attrVals : ∀ {c} (o : Obj c) (a : Attr c) → ⟦ attrType a ⟧
    δ        : ∀ {c} (o : Obj c) (a : Assoc c)
               → RVec (Obj (Δ a)) (assocRange a)
```

In Range

```
record InRange (i : Interval) : Set where
  open Interval i
  field
    value : ℕ
    {ni}  : True (decIsInRange i value)

RVec : (A : Set) (i : Interval) → Set
RVec A i = Σ (InRange i) (λ x → Vec A (value x))
```

- True x is $\top$ when x is true, and $\bot$ otherwise
- decIsInRange decides the IsInRange predicate
- Hence, if value is not in range, ni will have no inhabitant

Object Graph Example

Duck Family

Comments

- data types are only used for presentation's sake
- names should be built using the finite type Fin
- then additional properties can be automatically decided
- bidirectional associations are handled separately

Bidirectionality

```
AssocIx : ∀ {c} (o : Obj c) (a : Assoc c) → Set

IsBidirect : Bidirect G → Set
IsBidirect b = let open Bidirect b in
  ∀ (o : Obj class) (i : AssocIx o assoc) →
  let o' = lookup i (δ o assoc) in
  Σ [i' : AssocIx o' assoc'] (lookup i' (δ o' assoc') ≅ o)

record ObjDiagram {T} (G : ClassDiagram T) : Set₁ where
  open ClassDiagram G
  field
    objGraph : ObjGraph classGraph
  open ObjGraph objGraph
  field
    isBidirects : ∀ (i : Fin (proj₁ bidirects))
                  → IsBidirect (lookup i (proj₂ bidirects))
```

Conclusion

- we have implemented type-correct classes and objects in Agda
- dependent types provide a relatively elegant encoding
- these structures can be used to check and infer correct data
- Agda allows incremental development through meta-variables
- data and logic bound tightly together
- only basic constraints currently represented

Future Work

- implementation of OCL-style constraints
- representation of model transformations
- integration with Wolfram Kahl's graph transformation library
  - graph transformations through categoric DPOs/TGGs
  - requires sums, products, homomorphisms etc.
  - constraints also should be mapped
- attaching the implementation to an Eclipse frontend

Questions?