Applying Agda in Model Driven Design

Simon Foster

University of Sheffield, UK

joint work with J Derrick, O Rypáček[1], T Simons, G Struth

[1] King's College London

Model Driven Design

model driven architecture

- objective of formalising the software development process
- integration/evolution of heterogeneous system models
- model transformations relate models/system parts
- transformations can be
  - endogenous/exogenous
  - uni/bi-directional
- emphases
  - (semi-)automation
  - code generation
  - software life cycle management

Model Driven Design

opportunity for formal methods

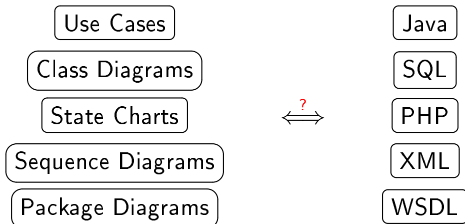
- formalise specification/transformations
- introduce formal development/verification techniques
- guarantee interoperability of heterogeneous models

Model Driven Design

this project

- formalise approach in dependent type theory
- develop transformation/refinement techniques
- construct provably correct transformations
- engineer provably consistent/coherent systems
- implement automated proof support

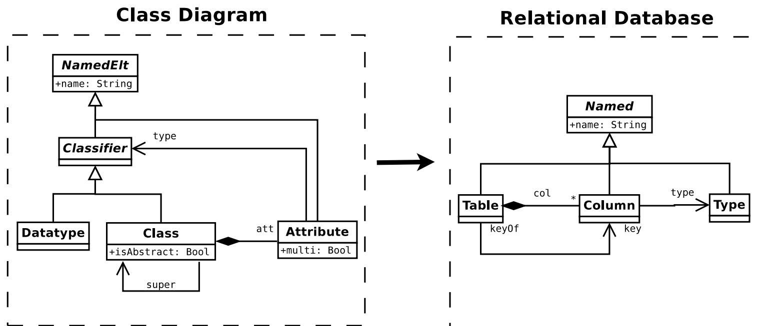
Model Transformations



basic idea

- models as relations/graphs
- model transformations as relations/graph transformations
- liberal approach to OMG MOF OCL BBD standards
- learn from languages like ATL, Kermeta, VIATRA2

Models as Graphs



model transformation

- link nodes/edges
- capture subclass hierarchy
- encode constraints on edges

Dependent Types for MDA

modelling

- specify (meta-)models via (co)inductive datatypes
- capture system constraints at type level

verification

- construct transformations while proving their correctness
- use proofs-as-programs for code generation

automation

- hide theory behind interface
- integrate ATP systems

ATP Integration

automated proof technology

- model checkers, SMT solvers, ATP systems, ...
- powerful proof search, limited expressivity
- based on classical logic, no program extraction

interactive proof technology

- expressive, but limited automation (traditionally)
- Coq: constructive, little automation, indirect prog. extraction
- Agda: "constructive" programming language, no automation
- PVS/Isabelle: integrated automation, but classical

what can Agda learn from Isabelle?

Agda

a dependently typed functional programming language

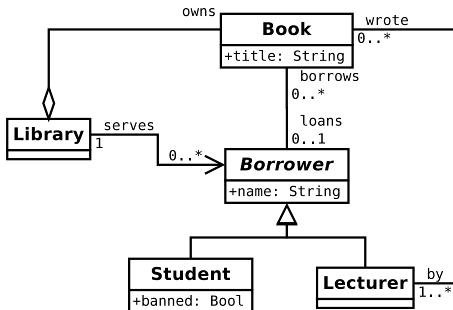
- Haskell-style syntax
- data types can depend on values (inductive families)
- parametrised modules

an interactive theorem prover

- based on Martin-Löf intuitionistic type theory
- program construction by metavariable refinement
- no formalisation gap between programs/proofs

...time for a demo!

Formalising MDA Concepts in Agda



create data types for

- meta level: building blocks for models
- object level: models
- models are graphs typed by the meta-model graph

Fundamental MDA Data Types

underlying structure

- Class Graphs
- Multiplicities
- Primitive Types
- Trees

meta level

- Class Diagrams (Types)
- Object Graphs (Values)

so far we don't model transformations!

Graphs in Agda

intuition

- a graph is represented as a function $\delta : V \times E \to V$
- for class graphs $V = Class$ and $E = Assoc$
- for object graphs $V = Obj$ and $E = Assoc \times Index$
- class graphs also give attribute schemas
- object graphs give attribute values
- dependent types link object graphs to class graphs

Class Graphs

```
record ClassGraph (τ : PrimTypes) : Set₁ where
  open PrimTypes τ
  field
    Class     : Setoid₀₀
    Attr      : IxSetoid₀₀ Class
    AttrTypes : ∃∃ Attr ⟶ Types
    Assoc     : IxSetoid₀₀ Class
    Subtype   : IxSetoid₀₀ Class
    Arity     : ∃∃ (Subtype ××I Assoc) ⟶ Multiplicity
    δ         : ∃∃ Assoc ⟶ Class
```

Primitive Types and Multiplicities

```
record PrimTypes : Set₁ where
  field
    Types : Setoid₀₀
    ⟦_⟧τ : Setoid.Carrier Types → Setoid₀₀
```

```
record Range : Set where
  field
    lb     : ℕ
    extent : Maybe ℕ
  ub = maybe′ (_+ℕ_ lb) lb extent
  InRange : ℕ → Set
  InRange n = lb ≤ℕ n × n ≤ℕ ub
  Index : Set
  Index = ∃ InRange
```

Object Graphs

```
record ObjGraph {τ} (C : ClassGraph τ) : Set₁ where
  open ClassGraph C

  field
    Obj       : IxSetoid₀₀ Class
    attrValue : ∀ {c} → ⌊ Obj ⌋I c → (a : ⌊ Attrs ⌋I c)
                  → ⟦ AttrTypes ⟨⟨$⟩⟩ a ⟧τ

    subtype   : Obj ↠ Subtype

    assocSize : ∀ {c} → (o : IxSetoid.Carrier Obj c)
                → (a : IxSetoid.Carrier Assoc c)
                → Multiplicity.Index (Arity ⟨⟨$⟩⟩ (subtype ⟨⟨$⟩⟩ o, a))

    Δ         : ∀ {c} → (o : IxSetoid.Carrier Obj c)
                  → (a : IxSetoid.Carrier Assoc c)
                  → (ar : Fin (proj₁ (assocSize o a)))
                  → IxSetoid.Carrier Obj (δ ⟨⟨$⟩⟩ a)
```

Class Diagrams

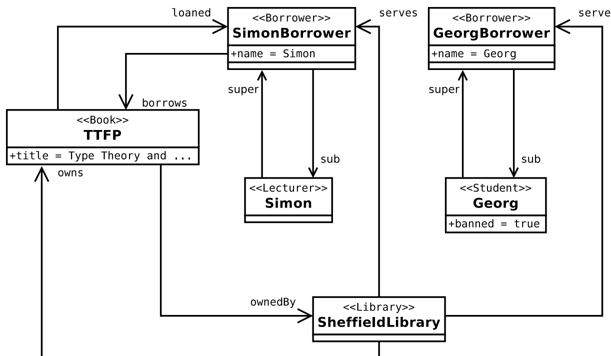
```
record ClassDiagram (τ : PrimTypes) : Set₁ where
  field
    classGraph : ClassGraph τ
  open ClassGraph classGraph

  field
    hierarchy  : Tree
    isAbstract : Path hierarchy → Bool
    positions  : Bijection Class (PropEq.setoid (Path hierarchy))

  data InherAssoc : Set where
    sub super : InherAssoc

  classDiagram : ClassGraph τ
  classDiagram = record {...}
```

Object Level Graph



such object graphs

- populate classes shown above
- provide the basis for transformations
- inheritance reduced to bidirectional associations at model level

How to represent transformations

programmed approach

- code to perform transformation written manually
- explicit constraints relate models
- semi-automatically generate code

graph grammar approach

- transformation is a set of pattern pairs
- rewrite engine applies grammar
- endogenous transformations as standard SPO/DPO
- exogenous transformations as triple graph grammar

Formalising MDA Concepts in Agda: Discussion

work so far

- infrastructure for implementing (meta-)*model graphs
- constraints can be formalised using modal formulae

ongoing work

- model specification infrastructure
  - basic ideas in place, realisation still fluid
  - categoric view needed to support graph grammars
- model transformations
  - design rule-based language
    (programmed / graph grammars)
  - integrate techniques from program transformation/refinement

ATP Integration

objectives

- make ATP power available for Agda
- increase MDA automation
- make MDA faster/more dependable

inspiration: Isabelle sledgehammer tactic

- relevance filter picks hypotheses for proof goal
- slegehammer calls external ATP/SMT systems
- external proofs reconstructed by Isabelle

can that be transferred to constructive context?

ATP Integration

our approach

- integration of equational ATP system Waldmeister

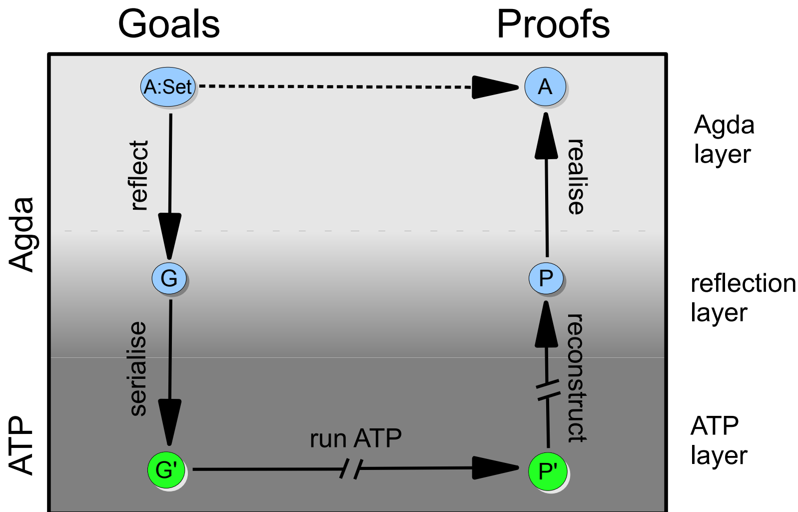
implementation

- reflection layer data types represent problems/proofs
- proof that rewrite sequences respect (propositional) equality
- Haskell module calls Waldmeister, reformats its proofs

design choices

- classical/constructive logic
- macro/micro-step proof reconstruction
- internal/external proof representation

ATP Integration



Proof Cycle Example

reflection layer axioms

- list signature Σ-List contains [], ::, ++ and rev
- "axiomatic class" for lists with reverse

$$
\begin{array}{lcl}
\text{`List} : \text{HypVec`List} & = & \text{HyVec } \Sigma\text{-List axioms } \textbf{where} \\
\text{++-nil} & = & \Gamma 1, \text{`[] `}{+\!\!+}\ \alpha \approx \alpha \\
\text{++-cons} & = & \Gamma 3, (\alpha\ \text{`::}\ \beta)\ \text{`}{+\!\!+}\ \gamma \approx \alpha\ \text{`::}\ (\beta\ \text{`}{+\!\!+}\ \gamma) \\
\text{rev-nil} & = & \Gamma 0, \text{`rev `[]} \approx \text{`[]} \\
\text{rev-cons} & = & \Gamma 2, \text{`rev}\ (\alpha\ \text{`::}\ \beta) \approx \text{`rev}\ \beta\ \text{`}{+\!\!+}\ (\alpha\ \text{`::}\ \text{`[]}) \\
\text{axioms} & = & (\text{++-nil :: ++-cons :: rev-nil :: rev-cons :: []})
\end{array}
$$

Proof Cycle Example

soundess/correctness of rewriting in Agda

- rewrite proof: $E, L, \Gamma \ \vdash \ A \Longrightarrow s \approx t$ with
  - hypotheses/axioms in E
  - additional lemmas in L
  - variable context Γ
  - assumptions A under Γ
  - terms s and t
- correctness of rewriting: (verified in Agda)

$$E, L, A \vdash s \approx t \iff [\![E]\!], [\![L]\!], [\![A]\!] \models [\![s]\!] \cong [\![t]\!]$$

- this forms the basis of proof reconstruction/serialisation

Proof Cycle Example

reflection layer proof goal for $\mathsf{rev} \circ \mathsf{rev} = \mathsf{id}$

$$
\begin{aligned}
&\mathsf{rev\text{-}rev\text{-}nil} : \text{`}\mathsf{List}, [\,], \Gamma0 \vdash [\,] \Rightarrow \text{`}\mathsf{rev}\ (\text{`}\mathsf{rev}\ \text{`}[\,]) \approx \text{`}[\,] \\
&\mathsf{rev\text{-}rev\text{-}nil} = \{\,\}\ 0
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{rev\text{-}rev\text{-}cons} :\ & \text{`}\mathsf{List}, ((\Gamma2,\ \text{`}\mathsf{rev}\ (\alpha\ \text{`}{+\!\!+}\ \beta) \approx \text{`}\mathsf{rev}\ \beta\ \text{`}{+\!\!+}\ \text{`}\mathsf{rev}\ \alpha) :: [\,]) \\
,\ & \Gamma2 \vdash ((\text{`}\mathsf{rev}\ (\text{`}\mathsf{rev}\ \beta) \approx \beta) :: [\,]) \\
\Rightarrow\ & (\text{`}\mathsf{rev}\ (\text{`}\mathsf{rev}\ (\alpha\ \text{`}{::}\ \beta))) \approx (\alpha\ \text{`}{::}\ \beta) \\
\mathsf{rev\text{-}rev\text{-}cons} =\ & \{\,\}\ 1
\end{aligned}
$$

Proof Cycle Example

Waldmeister proof input (induction step)

```
...

EQUATIONS        app(nil,xs) = xs
                 app(cons(x,xs),ys) = cons(x,app(xs,ys))
                 rev(nil) = nil
                 rev(cons(x,xs)) = app(rev(xs),cons(x,nil))
                 app(rev(ys),rev(xs)) = rev(app(xs,ys))
                 rev(rev(as)) = as
CONCLUSION       rev(rev(cons(a,as))) = cons(a,as)
```

Proof Cycle Example

Waldmeister proof output (induction step)

```
 Lemma 1: app(rev(x1),nil) = rev(x1)

   app(rev(x1),nil)
=    by Axiom 3 RL
   app(rev(x1),rev(nil))
=    by Axiom 5 LR
   rev(app(nil,x1))
=    by Axiom 1 LR
   rev(x1)

 Lemma 2: app(cons(x1,as),nil) = cons(x1,as)

 Lemma 3: rev(app(rev(as),cons(x1,nil))) = app(app(cons(x1,nil),as),nil)

 Lemma 4: rev(rev(cons(x1,as))) = cons(x1,as)

 Theorem 1: rev(rev(cons(a,as))) = cons(a,as)

   rev(rev(cons(a,as)))
=    by Lemma 4 LR
   cons(a,as)
```

Proof Cycle Example

reflection layer proof reconstruction (induction step)

$$
\begin{aligned}
\text{rev-rev-cons} \;:\;& \text{‘List}, ((\Gamma 2,\ \text{‘rev}\ (\alpha\ \text{‘}{+\!\!+}\ \beta) \approx \text{‘rev}\ \beta\ \text{‘}{+\!\!+}\ \text{‘rev}\ \alpha) :: [\,]) \\
,\;& \Gamma 2 \vdash ((\text{‘rev}\ (\text{‘rev}\ \beta) \approx \beta) :: [\,]) \\
\Rightarrow\;& (\text{‘rev}\ (\text{‘rev}\ (\alpha\ \text{‘}{::}\ \beta))) \approx (\alpha\ \text{‘}{::}\ \beta)
\end{aligned}
$$

rev-rev-cons $=$

$$
\begin{aligned}
&\text{fromJust (reconstruct ((inj}_1\ (\#\ 3), \text{true}, \text{eq-step}\ (0 ::^{l} [\,]^{\,l}) \\
&(\text{con}\ (\#\ 4)\ ([\,]^{\,x}) ::^{s} \text{con}\ (\#\ 5)\ ([\,]^{\,x}) ::^{s} [\,]^{\,s})) ::^{l} (\text{inj}_1\ (\#\ 0), \\
&\text{false}, \text{eq-step}\ (0 ::^{l} [\,]^{\,l})\ (\text{con}\ (\#\ 2)\ (\text{con}\ (\#\ 3)\ (\text{con}\ (\#\ 5) \\
&([\,]^{\,x}) ::^{x} [\,]^{\,x})::^{x} \text{con}\ (\#\ 1)\ (\text{con}\ (\#\ 4)\ ([\,]^{\,x}) ::^{x} \text{con}\ (\#\ 0)\ ([\,]^{\,x}) \\
&::^{x} [\,]^{\,x})::^{x} [\,]^{\,x})::^{s} [\,]^{\,s})) ::^{l} (\text{inj}_2\ (\#\ 0), \text{true}, \text{eq-step}\ ([\,]^{\,l}) \\
&(\text{con}\ (\#\ 0)\ ([\,]^{\,x}) \ldots
\end{aligned}
$$

ATP Integration: Discussion

contribution

- first (prototype) ATP integration into constructive ITP system
- microstep proof reconstruction
- important basis for program development in Agda

limitations

- infrastructure overhead negates ATP advantages
- only equational proofs supported
- proof reconstruction depressingly slow
  - reflection proof terms add overhead to proof normalisation
  - normalisation via shallow embedding

ATP Integration: Discussion

switch to a hybrid approach to proof reconstruction

- problem/proof formation by Haskell, proof checking by Agda
- easier to use, though less precise
- should provide a major speed increase
- may be wrong approach in the long term

Future Work

MDA implementation

- link our MDA data-types to RATH
- see if graph triples make sense in this context
- integrate program transformation and refinement techniques
  - graph grammars for semantics?
- build Eclipse tool with Agda as "hidden" proof engine

ATP integration

- optimise proof reconstruction
- integrate full FOL provers
- add counterexample generators/decision procedures

Further Information

publications

- S Foster, G Struth. Integrating an Automated Theorem Prover into Agda. NASAFM 2011
- S Foster, G Struth, T Weber. Automated Engineering of Relational and Algebraic Methods in Isabelle/HOL. RAMiCS 2011
- S Foster, O Rypáček, A Simons, G Struth. Model Transformation by Refinement in Constructive Logic. MELO 2011
- W Guttmann, G Struth, T Weber. Automating Algebraic Methods in Isabelle. ICFEM 2011
- W Guttmann, G Struth, T Weber. A Repository for Tarski-Kleene Algebras. ATE 2011

web site

http://simon-foster.staff.shef.ac.uk