Correctness of Object Oriented Models by Extended Type Inference

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

[1]Department of Computer Science, University of York
`simon.foster@york.ac.uk`

[2]Department of Computer Science, University of Sheffield
`{g.struth,o.rypacek}@dcs.shef.ac.uk`

Abstract. Modelling and analysing data dependencies and consistency between classes and objects is a complex task. We show that dependently typed programming languages can handle this in a particularly simple, convenient and highly automated way. Dependent datatypes are used to implement (meta)models for classes and objects directly and concisely. Data dependencies and similar system constraints are specified within the language's expressive type system. Verification and propagation of these constraints is handled by type inference, which can be enhanced by customised decision procedures or external solvers if needed. The approach thus supports the development of software models that are correct by construction.

1 Introduction

This research is motivated by the Model Driven Architecture[1] (MDA), where software is developed by integrating global platform independent system models with heterogeneous platform specific models. The MDA provides models for object-oriented designs at various meta-levels (MOF), languages for expressing constraints between models (OCL), and various languages and methods for model transformations. The approach depends on the ability to compose platform specific models in a highly automatic fashion, while maintaining global consistency and constraints that have been declared at the abstract level [9]. This is still difficult to achieve in practice.

This work focuses on modelling and analysing data dependencies and consistency between class diagrams, their corresponding object diagrams and meta-level templates that yield constraints for specifying these diagrams.

This work is inspired by and builds on previous work on formalising model transformations in constructive type theory [11]. We propose an approach that uses the dependently typed programming language Agda [2] for modelling system (meta)models for classes and objects, as they arise in MDA. Agda is a functional programming language similar to Haskell but with a much more powerful

[1] `http://www.omg.org/mda`

type system in which intuitionistic higher-order logic can be used for expressing type constraints. Since, in this context, type inference is undecidable, the programming language is also an interactive theorem prover (ITP), in which type constraints can be resolved by incrementally filling in holes in proofs. Agda has a similar theory pedigree to the Coq ITP [1].

At the specification side, dependently typed programming is very convenient for MDA because data dependencies between classes and objects can be declared very simply and directly by using dependent datatypes, in particular dependent records. At the verification side, data consistency can often be established fully automatically by type checking and type inference. For more complex type constraints, this can be augmented by customised decision procedures, domain specific solvers or automated theorem provers [4].

Our main contributions are as follows.

- We show how class graphs—a restricted version of the MOF—can be implemented by Agda's dependent records. As an example, we populate the general infrastructure for classes provided by a simple meta-model for classes.
- We demonstrate how object graphs—which are dependent on class graphs—can be implemented in such a way that class-level constraints, for instance range constraints of associations, are automatically maintained. As an example, we derive object diagrams that are correct by type checking with respect to the constraints imposed.
- To illustrate the use of more advanced constraints, we show how bidirectional associations can be declared and verified automatically.

These results show that, with Agda, modelling, verification and implementation of our approach can be achieved in one and the same language. This is in contrast to other approaches where external formal methods like B or verification approaches like model checking are used [13, 7].

Agda, as a functional language without side-effects, is also ideally suited for compositional system development. Compositionality is particularly important for integrating heterogeneous models in MDA.

While this paper focuses on the proof of concept that dependently typed programming has much to offer for MDA, and therefore shows and discusses the Agda implementation in detail, these technicalities should be hidden as far as possible to software engineers. A UML-style graphical toolkit providing templates for class and object diagrams could use our implementation as a backend, providing mechanisms for validating inputs and resolving system constraints behind the scenes in a highly sophisticated way.

The complete code featured in this paper can be found at our website[2].

2 Agda Preliminaries

This section contains a basic introduction to the features of Agda needed for this work. A more complete tutorial can be found on the Agda Wiki[3]. Agda is

[2] `http://www-users.cs.york.ac.uk/~simonf/MDA/`
[3] `http://wiki.portal.chalmers.se/agda`

a dependently typed functional programming language and proof assistant. Its syntax is inspired by Haskell, although Agda has support for both unicode and mixfix (e.g. ternary) operators. Agda is space sensitive – if a string has no spaces it is treated as a single name (or name part for a mixfix operator), for instance "x⩽y" is a single name whilst "x ⩽ y" consists of three names.

Agda is set apart as a programming language by its support for developing programs or systems that are *correct by construction*. Its type system is expressive enough to capture correctness properties concisely and sufficiently. These properties are verified within the development process by type checking. Whereas in simple cases this can be achieved fully automatically, more advanced tasks require interactive theorem proving within Agda. Agda also supports incremental development of programs and proofs by the technique of meta-variable refinement, which we illustrate below.

There are three main programming concepts which we here highlight:

- Algebraic Datatypes, which may be (co)inductive and can be used to define both data and propositions (e.g. proof datatypes);
- Functions, which can be recursive and contain proofs;
- Dependent records, a special form of datatype with field projections.

Datatypes are specified using a type constructor declaration, followed by a list of constructors with their respective types.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _::_ : {n : ℕ} → (x : A) → (xs : Vec A n) → Vec A (suc n)
```

Set is the type of all types. The natural numbers datatype declares the two usual constructors. The vector type Vec is a *dependent* datatype, because it is parameterised by its length, a property that depends on the particular data. The type constructor has two parameters. The first parameter, A, is an arbitrary element of Set which is fixed for all constructors. This is indicated by its position before the colon. The second parameter is a natural number which specifies the vector's length, and varies depending on the constructor. The first constructor [] constructs a Vec over A with zero length. The second infix constructor ::, given a value of type A and a vector of length n, constructs a vector of length suc n. Parameter n is hidden by the presence of braces. This means that the type system will try to infer its value via the other parameters and thus it need not be explicitly given. Parameter types in general can often also be inferred by insertion of a ∀ quantifier. Hidden arguments are useful to reduce verbosity of declarations. So 1 :: 2 :: 3 :: [] is a vector of length three over ℕ.

Functions in Agda are total; a coverage checker ensures that all possible inputs have associated outputs. Partial functions can be specified by a partiality type, akin to Haskell's Maybe. Functions are specified using a type-signature followed by a sequence of equations. We define function ++ and map below.

```
_++_ : {A : Set} {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
[] ++ ys        = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

map : {A B : Set} {m : ℕ} → (A → B) → Vec A m → Vec B m
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Concatenation, ++, is a binary operation on vectors defined, as usual, by recursion. It takes two vectors of element type A with lengths m and n and produces a vector of length m + n. This last property is beyond the expressivity of most traditional type systems. The map function takes a vector of A's and a function from A to B and produces a vector of B's by applying the function to every element. It is defined in the usual way.

Functions can also be used to represent lemmas and proofs; here the property that ++ distributes over map. We prove it by meta-variable refinement:

```
map-++ : ∀ {m n} {A B : Set} (f : A → B) (xs : Vec A m) (ys : Vec A n)
         → map f xs ++ map f ys ≡ map f (xs ++ ys)
map-++ f xs ys = { } 0
```

We begin with a type declaration, which encodes the proof goal for a function f and arbitrary vectors xs and ys, all supplied as parameters. In the type declaration, ≡ represents propositional equality, that is equality of normal forms in all contexts. The function equation is populated with a meta-variable hole, { } 0, which we incrementally fill in using a divide and conquer strategy. To proceed with the inductive proof we have Agda split xs into its two possible cases.

```
map-++ f [] ys        = { } 0
map-++ f (x :: xs) ys = { } 1
```

We can then fill in both of these proof obligations as below.

```
map-++ f [] ys        = refl
map-++ f (x :: xs) ys = cong (_::_ (f x)) (map-++ f xs ys)
```

The first case has type map f [] ++ map f ys ≡ map f ([] ++ ys), after substitution, which normalises to map f ys ≡ map f ys. Therefore the two terms are propositionally equal by application of the reflexivity constructor refl. The second case is obtained by application of the induction hypothesis and the congruence rule cong under the context $C[X] = (\mathsf{f\ x}) :: X$.

Records are specified in the usual way by listing fields with their types.

```
record Person : Set where
  field
    name       : String
    age        : ℕ
    ageInRange : age < 200
```

This is a *dependent* record because fields can depend on predecessors. Here the field called ageInRange, which is a proof of the proposition age < 200, depends on age. Thus fields can be used to encode both data and data constraints. We make much use of this for our MDA datatypes.

3 Overview of the Encoding

The fundamental MDA datatypes are UML class diagrams and object diagrams. To deal with them uniformly, MDA provides the meta-object facility (MOF) in which templates for objects are provided as class diagrams. Here we do not attempt an accurate portrayal of the MOF, but rather a simplified interpretation in terms of directed graphs (digraphs).

We model digraphs as labelled transition systems, using a transition function to map vertices and edges onto vertices. Recall that given a set $\mathcal{V}$ of vertices and $\mathcal{E}$ of edges, a digraph is a function $\delta : \mathcal{V} \times \mathcal{E} \to \mathcal{V}$.

- In Section 4 we introduce a set of class graphs, ClassGraph : Set_1 in Agda. A ClassGraph can be regarded as a digraph where the set of vertices is Class and the transition function is $\mathit{\Delta}$: ∀ {c : Class} → Assoc c → Class.
- In Section 6 we define ObjGraph : ClassGraph → Set, assigning to each class graph c : ClassGraph a set of all object graphs validating c. ObjGraph is a digraph indexed by Class and the transition function is δ followed by a projection out of the object vector.
- In Section 8 we add bidirectionality constraints to class and object graphs.

The intermediate sections illustrate these implementations with simple examples.

4 Class Graphs in Agda

In Agda we represent class graphs as a dependent record.

```
record ClassGraph (Types : Set) : Set₁ where
  field
    Class      : Set
    Attr       : Class → Set
    Assoc      : Class → Set
    attrType   : {c : Class} → Attr c → Types
    assocRange : {c : Class} → Assoc c → Interval
    Δ          : {c : Class} → Assoc c → Class
```

A class graph is parameterised over a set Types of primitive types, which we use in class attributes. Class is the set of class names. Attr and Assoc are parametric types, indicated by their definition as functions from Class into Set. They give the attribute and association names for each class. Field attrType gives a type to each attribute, and assocRange a target multiplicity for each association. Finally, $\mathit{\Delta}$ gives the target class of each association. This record is dependent because, for instance, assocRange depends on the definition given for Assoc. Constraints imposed on Assoc are therefore automatically inherited by assocRange.

Specifications like this would be difficult to formulate as concisely in languages without dependent types. We strongly use dependent records for the implementation of MDA concepts in this paper.

The parameter Interval in the definition of assocRange is defined as follows.

```
record Interval : Set where
  constructor _–_
  field
    lb : ℕ
    ub : Maybe ℕ
  IsInRange : ℕ → Set
  IsInRange n = lb ℕ≤ n × n ≤ⁱ ub
```

Interval ranges over the natural numbers. It consists of a lower bound lb and an upper bound ub, with just n representing upper bound n and nothing an unbounded range $*$. The predicate IsInRange over $\mathbb{N}$ defines whether the given number is in range by means of a pair of inequalities, which are customised for the respective types. In constructive logic $\times$ performs the function of $\wedge$. We also define a simple binary constructor so that we can write ranges like 0 – just 1.

5 A Class Graph Example

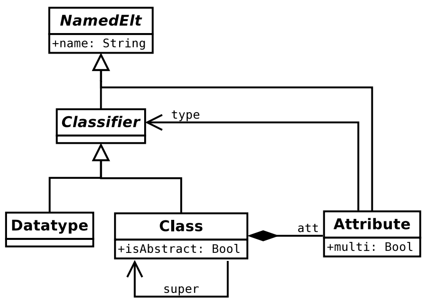

Fig. 1. Basic metamodel class diagram

To exemplify the use of class graphs, we populate a simple meta-model for a class diagram as shown in Figure 1. We first declare some datatypes to act as names for classes, associations and attributes.

```
data CDClass : Set where
  NamedElt Classifier Class Attribute DataType : CDClass
data CDAssoc : CDClass → Set where
  att super   : CDAssoc Class
  type owner  : CDAssoc Attribute
  *super      : ∀ {c} → CDAssoc c
data CDAttr : CDClass → Set where
  name         : CDAttr NamedElt
  isAbstract   : CDAttr Class
  multivalued  : CDAttr Attribute
```

The classes in this diagram are the labels of the nodes in the class graph. In our case these are precisely the elements NamedElt, Classifier, Class, Attribute and DataType of the datatype CDClass. The elements CDAssoc and CDAttr represent the names of associations and attributes in the class graph. They are parameterised by the name of the class from which they are drawn.

CDAssoc has elements att—the attributes of each class, type—the attribute type, owner—the owner class of an attribute, super—the superclass of a class, and *super—a polymorphic association which we use to model inheritance at the meta-level. We could automatically instrument the representation of inheritance through a preorder on the classes, but for this simple example it will be encoded manually. CDAttr has elements name, isAbstract and multivalued, as depicted in the example diagram.

To assign types to each attribute we also need a set Prim of primitive types. The types $\mathbb{N}$, Bool and String are all drawn from Agda's standard library[4].

```
data Prim : Set where Nat Bl Str : Prim
⟦_⟧ : Prim → Set
⟦ Nat ⟧ = ℕ
⟦ Bl ⟧  = Bool
⟦ Str ⟧ = String
```

We now define the functions for attrType, assocRange and Δ for the remaining fields of our instance of ClassGraph.

```
cd-attrType : ∀ {c} → CDAttr c → Prim
cd-attrType name        = Str
cd-attrType isAbstract  = Bl
cd-attrType multivalued = Bl

cd-asrn : ∀ {c} → CDAssoc c → Interval
cd-asrn {NamedElt} *super = none
cd-asrn {_} *super        = one
cd-asrn {Class} att       = many
cd-asrn {Class} super     = optional
cd-asrn {Attribute} type  = one
cd-asrn {Attribute} owner = one
```

```
cd-Δ : ∀ {c} → CDAssoc c → CDClass
cd-Δ {NamedElt} *super  = NamedElt
cd-Δ {Class} att        = Attribute
cd-Δ {Class} super      = Class
cd-Δ {Class} *super     = Classifier
cd-Δ {Attribute} type   = Classifier
cd-Δ {Attribute} owner  = Class
cd-Δ {Attribute} *super = NamedElt
cd-Δ {DataType} *super  = Classifier
cd-Δ {Classifier} *super = NamedElt
```

The definition of cd-attrType assigns to each attribute in the class graph a type symbol from Prim. The function cd-asrn associates a range to each association, where none is empty, many is $0 - *$, optional is $0 - 1$ and one is $1 - 1$. The implementation of these functions is not shown. We must pattern match on both the class name and association name because of the polymorphic *super which has a different arity for each type. In particular for NamedElt it is empty as there is no superclass. The code for Δ essentially programs the transition function of the graph, that is, it gives a target class to each association. In particular *super targets the direct superclass of each class.

Using these functions we can complete the definition of our class graph.

[4] http://wiki.portal.chalmers.se/agda/agda.php?n=Libraries.StandardLibrary

```
classGraph : ClassGraph Prim
classGraph = record
  { Class      = CDClass  ; Attr     = CDAttr
  ; Assoc      = CDAssoc  ; attrType = cd-attrType
  ; assocRange = cd-asrn  ; Δ        = cd-Δ }
```

6 Object Graphs in Agda

We can now define object graphs. At this level we need to provide an infrastructure for mapping the fields in an object graph, which represent the object instance data, to fields in a class graph, which represent their meta information.

```
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 ⟧
    assocIndices : ∀ {c} (o : Obj c) (a : Assoc c)
                   → InRange (assocRange a)
    δ            : ∀ {c} (o : Obj c) (a : Assoc c)
                   → Vec (Obj (Δ a)) (value (assocIndices o a))
```

An object graph is parametrised by a class graph G, from which its structure is drawn, and a primitive type interpretation function ⟦_⟧, which assigns an Agda type to each abstract type. Within the record we first open the class graph G, thus bringing its fields into scope. Obj defines the set of all objects for each class, thus an Obj c is an object of class c. Field attrVals assigns, for each object and attribute name, a value for the attribute, using ⟦_⟧ to give a concrete type. Field assocIndices gives a cardinality for each association of each object, which naturally must satisfy the constraint of being in the range of the association's interval, as specified in G. Finally, δ gives, for each object and association, a vector of target objects of the association's target class (Δ a) and given size.

ObjGraph is again heavily dependent, so for example all fields depend on the definition of the underlying class graph, and, in addition, attrVals, assocIndices and δ depend on the definition of Obj. From the object oriented point of view this is, of course, obvious, but being able to encode this directly in a specification language is certainly unusual (without dependent types).

Values within intervals are provided by the type InRange.

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

InRange uses a decision procedure to check whether a number is in the range of a given interval. This is, again, declared at the type level. It shows how Agda's type system can be effectively augmented with customised decision procedures. The InRange record consists of a number value and a proof that this value is in the correct range. The proof is automatically provided by the decision procedure

```
decIsInRange : (i : Interval) (n : ℕ) → Dec (IsInRange i n)
```

which makes direct use of the Agda standard library decision procedure for $\leqslant$.

The type Dec P represents a decision of the proposition P, containing either yes P or no ¬P. True is a function which returns type $\top$ if the parameterised value is yes and otherwise $\bot$. In the context of Agda, $\bot$ is a vacuous (unsatisfiable) type. $\top$ is a single element type which can also be automatically populated by the type checker. This means that field ni can be automatically inferred if value is populated with a number in the given range, whilst an out-of-range value will not yield a proof and invalidate InRange. We can therefore write $\#^{\mathsf{r}}$ 1 : InRange (0 – just 1), which is certified to be correct automatically by the type checker. Conversely, $\#^{\mathsf{r}}$ 6 : InRange (1 – just 5) will not type check as there is no proof of 6 $\leqslant^{\mathsf{i}}$ just 5. An IsInRange proof witness can then be extracted from a valid InRange and used in other proofs.

This simple example illustrates how constraints on objects and classes can effectively be captured by extended type checking in Agda, based on decision procedures, automated theorem provers or other solvers.

7 Populating Correct Object Graphs

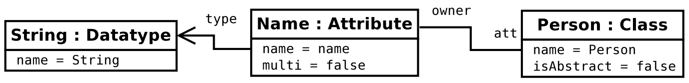

Fig. 2. Basic model object diagram

This section shows how the class and object graph infrastructure developed in the previous sections can be used for populating concrete object graphs while checking the constraints imposed on classes and objects at the type level. This process can be seen in analogy to completing a template or filling in a web form while type checking is used behind the scenes to guarantee that the data is correct. However, while usually only simple properties can be verified, for instance that some input string consists of numbers, much more powerful properties and system constraints can be captured by Agda's expressive type system.

As an example, we construct the object graph of Figure 2. Agda supports type-safe incremental construction of data through meta-variable refinement, which we use to define the object graph. We first create datatypes to represent our objects.

```
data PersonObj : CDClass → Set where
  Person   : PersonObj Class
  Person′  : PersonObj Classifier
  Person″  : PersonObj NamedElt
  Name     : PersonObj Attribute
  Name′    : PersonObj NamedElt
  Str      : PersonObj DataType
  Str′     : PersonObj Classifier
  Str″     : PersonObj NamedElt
```

Each object requires identifiers for each of the superclasses so that the respective *super associations can be populated. In real applications, such names would be allocated automatically; they are here shown explicitly for the sake of presentation. Using the object type we can then go ahead to construct the three functions attrVals, assocIndices and δ, each by meta-variable refinement. We now show this step-by-step. First we create the template for our object graph.

```
personOG : ObjGraph classGraph ⟦_⟧
personOG = record
  {Obj           = PersonObj
  ; attrVals     = og-attrVals
  ; assocIndices = og-assocIndices
  ; δ            = og-δ}
og-attrVals : ∀ {c} → PersonObj c → (a : CDAttr c) → ⟦ attrType a ⟧
og-attrVals o a = { } 0
og-assocIndices : ∀ {c} → PersonObj c → (a : CDAssoc c)
                  → InRange (assocRange a)
og-assocIndices o a = { } 1
og-δ : ∀ {c : CDClass} (o : PersonObj c) (a : CDAssoc c)
       → Vec (PersonObj (Δ a)) (value (og-assocIndices o a))
og-δ o a = { } 2
```

The types of these three functions can be inferred automatically, but we write them explicitly for the sake of explanation. The body of each function is populated by a meta-variable hole. To fill in og-attrVals we case split on first the o parameter, the type of objects, and then the a parameter, the attribute type. This gives us the following holes to fill in (with corresponding types).

```
og-attrVals Person isAbstract   = { } 0  -- Bool
og-attrVals Person″ name        = { } 1  -- String
og-attrVals Name multivalued    = { } 2  -- Bool
og-attrVals Name′ name          = { } 3  -- String
og-attrVals Str″ name           = { } 4  -- String
```

Agda infers the type of each meta-variable and presents this information to the user. We have added comments that indicate these types. These meta-variables can all be filled in with correct values, and the type checker prevents us from substituting an incorrectly typed value.

$$\begin{aligned}
&\mathsf{og\text{-}attrVals\ Person\ isAbstract} &&= \mathsf{false}\\
&\mathsf{og\text{-}attrVals\ Person''\ name} &&= \texttt{"Person"}\\
&\mathsf{og\text{-}attrVals\ Name\ multivalued} &&= \mathsf{false}\\
&\mathsf{og\text{-}attrVals\ Name'\ name} &&= \texttt{"name"}\\
&\mathsf{og\text{-}attrVals\ Str''\ name} &&= \texttt{"String"}
\end{aligned}$$

We follow a similar procedure for og-assocIndices, each case of which needs to be populated by a suitable InRange for the corresponding association. For instance the case for association owner of Name has range one $(1-1)$, meaning the only possible value is 1.

$$\mathsf{og\text{-}assocIndices\ Name\ owner} \ = \ \#^{\mathsf{r}}\ \mathsf{1}$$

If we try and insert any other value, for instance

$$\mathsf{og\text{-}assocIndices\ Name\ owner} \ = \ \#^{\mathsf{r}}\ \mathsf{2}$$

the type checker will fail to resolve the internal constraint proof (ni) of the InRange, since it resolves to $\bot$. In this way the type system acts as a static checker for the multiplicities whilst we build the diagram.

Finally we construct the object assocation function og-δ, each element of which is a vector of suitably typed object, with length drawn from the corresponding association index.

$$\begin{aligned}
&\mathsf{og\text{-}\delta\ Person\ att} &&= \{\,\}\ \mathsf{0} && \text{-- Vec (PersonObj Attribute) 1}\\
&\mathsf{og\text{-}\delta\ Person\ super} &&= \{\,\}\ \mathsf{1} && \text{-- Vec (PersonObj Class) 0}\\
&\mathsf{og\text{-}\delta\ Person\ {*}super} &&= \{\,\}\ \mathsf{2} && \text{-- Vec (PersonObj Classifier) 1}\\
&\mathsf{og\text{-}\delta\ Person'\ {*}super} &&= \{\,\}\ \mathsf{3} && \text{-- Vec (PersonObj NamedElt) 1}\\
&\mathsf{og\text{-}\delta\ Person''\ {*}super} &&= \{\,\}\ \mathsf{4} && \text{-- Vec (PersonObj NamedElt) 0}\\
&\ldots
\end{aligned}$$

We then fill in each meta-variable with associations from Figure 2.

$$\begin{array}{lll@{\qquad}lll}
\mathsf{og\text{-}\delta\ Person\ att} & = & [\mathsf{Name}] & \mathsf{og\text{-}\delta\ Name\ owner} & = & [\mathsf{Person}]\\
\mathsf{og\text{-}\delta\ Person\ super} & = & [\,] & \mathsf{og\text{-}\delta\ Name\ {*}super} & = & [\mathsf{Name'}]\\
\mathsf{og\text{-}\delta\ Person\ {*}super} & = & [\mathsf{Person'}] & \mathsf{og\text{-}\delta\ Name'\ {*}super} & = & [\,]\\
\mathsf{og\text{-}\delta\ Person'\ {*}super} & = & [\mathsf{Person''}] & \mathsf{og\text{-}\delta\ Str\ {*}super} & = & [\mathsf{Str'}]\\
\mathsf{og\text{-}\delta\ Person''\ {*}super} & = & [\,] & \mathsf{og\text{-}\delta\ Str'\ {*}super} & = & [\mathsf{Str''}]\\
\mathsf{og\text{-}\delta\ Name\ type} & = & [\mathsf{Str'}] & \mathsf{og\text{-}\delta\ Str''\ {*}super} & = & [\,]
\end{array}$$

This simple example shows how class and object graphs can be generated that are correct by construction. We are using the type system to check quite complicated constraints about the structures, rather than simple set-style constraints. Already simple object graphs can capture rather complex data dependencies that even involve the corresponding class. In a dependently type language, resolving these dependencies does not require any external proof engines or solvers. The entire process in this section is completely automatic as far as type inference is concerned. For instance, it is inferred that object Person has type Obj Class, class Class has association att, and association att in the class graph has range many and it must point to objects of type Attribute from the definition of Δ. It therefore follows that the Person case of function δ must consist of a vector of Obj Attribute, of a length within the many interval.

Although the constraints in our small examples are relatively simple, more complex constraints can be encoded at the type level if they can be expressed in intuitionistic higher-order logic.

8 Type Checking Bidirectionality Constraints

This section gives an example of how the verification of more complex constraints can be based on type checking. Many associations in class diagrams are bidirectional, for instance in Figure 1 att and owner are bidirectional. In our class graph model we encode a bidirectional association as a pair of unidirectional associations, but so far cannot explicitly declare and verify the constraint that a given association is bidirectional. We encode bidirectionality of associations with the following record.

```
record Bidirect {T} (G : ClassGraph T) : Set₁ where
  constructor [_⟷_]_
  open ClassGraph G
  field
    {class}  : Class
    assoc    : Assoc class
    assoc'   : Assoc (Δ assoc)
    converse : Δ assoc' ≡ class
```

Bidirect consists of a pair of associations, the first of which has its source in class and its target in the associated class, whereas for the second association source and target class are swapped, which is ensured by the converse constraint. We then use this new record to extend class graphs to class diagrams with bidirectional associations.

```
record ClassDiagram (Types : Set) : Set₁ where
  field
    classGraph : ClassGraph Types
    bidirects  : Σ [n : ℕ] (Vec (Bidirect classGraph) n)
```

A class diagram simply adds a vector of associations which are bidirectional. These can then be used in object diagrams, which constrain bidirectionality of concrete associations. First we define the interpretation function for bidirectional constraints with the function IsBidirect.

```
AssocIx : ∀ {c} (o : Obj c) (a : Assoc c) → Set
AssocIx o a = Fin (value (assocIndices o a))

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)
```

A bidirectional relation is encoded between associations assoc and assoc′ as a logical formula that satisfies the following condition. For every object o of the source class of assoc, and each of its indices i, targeting object o′, there exists an index i′ of assoc′ targeting object o. To satisfy this constraint for a given object graph we have to supply a lambda term of the correct type. In certain circumstances such a lambda term can be generated automatically.

We can then enforce this constraint when building object diagrams.

```
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))
```

An object diagram consists of an object graph and a function mapping each bidirectional association to a proof that bidirectionality holds.

9 A Bidirectionality Example

We can extend our examples from the previous sections, making att and owner converses of each other. This will turn our type into an accurate representation of the class diagram in Figure 1, ensuring that only valid object diagrams are elements.

```
classDiagram : ClassDiagram Prim
classDiagram = record
  {classGraph = classGraph
  ; bidirects = (2, [att ⟷ owner] PropEq.refl :: [owner ⟷ att] PropEq.refl :: [])}
```

Both directions must be stated. Each constraint also uses the simple proof PropEq.refl that following att and then owner returns to Class, and vice versa. We can then extend our object graph to an object diagram.

```
personOD : ObjDiagram classDiagram
personOD = record
  {objGraph = personOG
  ; isBidirects = isBidirects}
  where open ObjGraph personOG
    isBidirects : (i : Fin (proj₁ bidirects)) → IsBidirect (lookup i (proj₂ bidirects))
    isBidirects zero Person zero = zero, HetEq.refl
    isBidirects (suc zero) Name zero = zero, HetEq.refl
```

The function isBidirects satisfies both of the bidirectional constraints (vacuous cases are omitted). The first requires a proof of

```
Σ [i : Fin 1] (lookup i (Person :: []) ≅ Person)
```

where i is an index of owner under object Name, which is reached by following att from Person. This is trivially satisfied by index zero, with Person ≅ Person satisfied by HetEq.refl. The second case is similarly satisfied, but with att and owner reversed. Both of these proofs can be automatically discharged by Agda's auto tactic. This completes the definition of the object diagram.

10 Related Work

The area of formal semantics for UML is vast and space restricts their full consideration, though we note that both Object-Z and VDM have class diagram mappings. Specific to our work is the use of graphs to represent class diagrams, for which a comprehensive discussion exists in [8]. We use a fairly standard graph encoding, though with the addition of dependent types.

Works with similar aims using different ITPs exist. For instance, HOL-OCL [3] is a well-developed Isabelle/HOL library for verifying OCL expressions on UML class diagrams. Similarly, Object-Z has been mechanised [12]. The key difference in our work is the intimate relationship between data and proof, provided by dependent type theory. Nevertheless, Agda lacks the automated proof support which more mature ITPs (like Isabelle) enjoy, though similar results could potentially be achieved [4]. Moreover, we do not currently support OCL, but since we do have well-typed navigation an elegant implementation is possible.

11 Conclusion and Future Work

We have implemented basic MDA concepts and shown how Agda can be used for inferring system constraints. With dependent types, system dependencies and constraints can be modelled succinctly and directly; they can be resolved, often automatically, by extended type inference.

Additional work focuses on the implementation of a library for representing valid models and transformations between them. In general a model transformation can be represented as a function.

```
_⇒_ : ∀ {T} → ClassDiagram T → ClassDiagram T → Set
P ⇒ Q = ObjDiagram P → ObjDiagram Q
```

Such a function must, for each valid instance of the class diagram P, provide a valid instance of the class diagram Q. We can use the type system to aid with construction of such a transformation by having it exhaustively supply all possibilities for the input, thus ensuring that a model transformation is complete.

Refinement and implementation of transformations can proceed in several ways. We are currently collaborating on extending an Agda based graph transformation library [6] for this purpose. Graph transformations are one of the standard approaches to model transformations. Alternatively we are designing a state-based embedded language in Agda for graph traversals and manipulation, inspired by languages such as ATL [5] and Kermeta [10]. The type system can be used during program construction to inform programmers about data and constraints relevant to a particular object. Also, integration of automated theorem provers in Agda beyond the current prototype [4] would be of great benefit to the semi-automatic composition and development of model transformations.

In conclusion, we believe that Agda provides many benefits to the integration of formal methods, for instance by both ensuring that code is correct with respect to a suitable model and supplying useful information during code construction. In applicable formal methods the Agda layer needs to be hidden as much as possible behind an interface, for which a high degree of automation is a prerequisite. In the future we would therefore like to see an Eclipse frontend, which would interface to a suitable Agda domain-specific language and convert Agda error messages and information into a format readable by a software engineer.

Acknowledgements. The authors acknowledge support for this project by the EPSRC (grants EP/G031711/1 and EP/G031711/1).

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science, Springer (2004)
2. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda - a functional language with dependent types. In: TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer (2009)
3. Brucker, A., Wolff, B.: HOL-OCL: Experiences, consequences and design choices. In: UML '02. LNCS, vol. 2460, pp. 196–211. Springer (2002)
4. Foster, S., Struth, G.: Integrating an automated theorem prover into Agda. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods 2011. LNCS, vol. 6617. Springer (2011)
5. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
6. Kahl, W.: Dependently-typed formalisation of typed term graphs. In: Echahed, R. (ed.) TERMGRAPH 2011. pp. 38–53. EPTCS (2011)
7. Knapp, A., Merz, S.: Model checking and code generation for UML state machines and collaborations. Tech. Rep. 2002-11, Institut für Informatik, Universität Augsburg (2002), in Proc. FM-TOOLS 2002
8. Kuske, S., Gogolla, M., Kreowski, H., Ziemann, P.: Towards an integrated graph-based semantics for UML. Software and Systems Modeling 8, 403–422 (2009)
9. Lano, K.: Constraint-driven development. Information & Software Technology 50(5), 406–423 (2008)
10. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer (2005)
11. Poernomo, I.: Proofs-as-model-transformations. In: ICMT '08. LNCS, vol. 5063, pp. 214–228. Springer (2008)
12. Smith, G., Kammüller, F., Santen, T.: Encoding Object-Z in Isabelle/HOL. In: ZB '02. LNCS, vol. 2272, pp. 82–99. Springer (2002)
13. Snook, C.F., Butler, M.J.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (2006)