Model Transformation by Refinement in Constructive Logic

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

[1]Department of Computer Science, University of Sheffield
{s.foster,a.simons,g.struth}@dcs.shef.ac.uk

[2]Department of Informatics, King's College London
ondrej.rypacek@kcl.ac.uk

Abstract. We present first steps of a formalisation of meta modelling in a constructively typed programming language, explaining its potential for specifying model transformations. We describe our meta-model encoding and outline our automated theorem prover integration. Our aim is to provide an environment for formally developing software transformations, which are correct by construction and machine-checked proof.

1 Introduction

Model transformations define functions between meta-models, for instance, the mapping of class diagrams to corresponding relational databases. Such transformations are usually described through a programming task in a model transformation language. Our desire is to develop a more formal approach, in which model transformations are constructed by *refinement* from a logical specification. Traditional logical specification languages experience what can be called a "formalisation gap", in that the declarative logic specifying the program is different from the implementation language. However, in *constructive logic* (or type theory) specification, program and correctness proofs can be obtained in one and the same language, because proofs *are* programs.

In particular, the language *Agda* is both a theorem prover *and* a functional programming language, such that program development and correctness proofs become one and the same activity. This is of particular interest in the *Model Driven Architecture* where heterogeneous formalisms from MOF and UML can be used to describe a system. Once a formal semantics of MOF has been implemented, it becomes to possible to view a model transformation as a function

$$\mathcal{M}_1 \times \mathcal{C}_1 \xrightarrow{\mathcal{D}} \mathcal{M}_2 \times \mathcal{C}_2$$

where $\mathcal{M}_n$ is a meta-model, $\mathcal{C}_n$ a set of constraints on the meta-model and $\mathcal{D}$ a set of constraints which specifies the transformation.

Furthermore, since program construction in Agda is performed principally through *meta-variable refinement* where a program is built in a "divide and conquer" approach, it is clear to see what the remaining proof obligations of a partially constructed transformation are. This allows a form of *semi-automated* incremental programming where Agda aids in gradually zeroing-in on suitable values to fill holes in a transformation.

However, Agda currently provides only limited automation of proof search. This process can drastically be improved by the use of *automated theorem provers* (ATPs). An ATP is a program for proving first-order logic problems and we have provided a prototypical integration of such a prover into Agda.

The main benefit of our approach is the clean integration of logical/declarative and procedural aspects of model transformations and a systematic incremental approach to their construction and optimisation. In practical applications the Agda language and its proof facilities will, to a large extent, be hidden behind an interface. ATPs also free the users from manually satisfying trivial constraints and allows them to focus on the more conceptual aspects.

In this paper we outline our current work on meta-modelling and ATP integration in Agda. While this illustrates the main features of the approach, its applications and further development are left for future work.

2 Agda

The Agda tool [1] is a constructively typed functional programming language and at the same time a proof-assistant. This section yields a brief introduction. The following inductive data-type declaration introduces the natural numbers.

```
data ℕ : Set where
  zero : ℕ
  suc : (n : ℕ) → ℕ
```

Data-types can also be used to define logical propositions and specifications.

```
data _⩽_ : ℕ → ℕ → Set where
  z⩽n : ∀ {n} → zero ⩽ n
  s⩽s : ∀ {m n} (m⩽n : m ⩽ n) → suc m ⩽ suc n
```

The elements of this data-type are *inductive proofs* of $\leqslant$. For instance, s⩽s z⩽n is a proof of $1 \leqslant 2$. One can also define n < m as suc n ⩽ m. Functions, such as addition or multiplication in $\mathbb{N}$, can be defined as usual. In addition, functions can also represent proofs of propositions. For instance, one can write

```
greater : ∀ (n : ℕ) → ∃ (λ (m : ℕ) → n < m)
greater n = ?
```

This is a "specification" that for every natural number there exists a greater natural number. The question mark informs Agda that we wish to construct the proof by *meta-variable refinement.* Agda provides tactics for decomposing a proof goal into proof templates in which meta-variables indicate the holes that need to be filled by the programmer. Proving this fact amounts to constructing a function, a program which implements the specification. In this case, we perform a *case-split* on parameter n with respect to the constructors. This yields the proof goals greater zero $=$ { }0 and greater (suc n) $=$ { }1. The first requires a value of type $\exists(\lambda\ \mathsf{m} \longrightarrow 0 < \mathsf{m})$ and the second a value of type $\exists(\lambda\ \mathsf{m} \rightarrow \mathsf{suc\ n} < \mathsf{m})$, assuming $\exists(\lambda\ \mathsf{m} \rightarrow \mathsf{n} < \mathsf{m})$. Further refinements allow the programmer to fill in both holes and complete the proof:

```
greater zero    =  1, s≤s z≤n
greater (suc n) =  (suc (proj₁ (greater n)), s≤s (proj₂ (greater n))
```

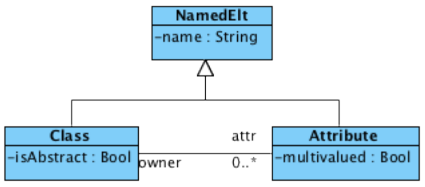
The syntactic details should not concern us at this point. This proof style lends itself naturally to incremental program construction, where writing a program and proving its correctness are one and the same activity.

3 Meta-modelling in Agda

A meta-model in Agda is defined as the following record type.

```
record MetaModel : Set₁ where
  field
    Cls : Set
    Attrs Assocs : Cls → Set
    AttrTys : {i : Cls} → Attrs i → PrimTypes
    AssocTys : {i : Cls} → Assocs i → Cls
    multis : {i : Cls} → Assocs i → ℕ x Maybe ℕ
```

Records in Agda are *dependent* – each line can depend on those defined before. Here `Cls` is a set of *class names*, which is equipped with a partial order defining the subclass relationship (not shown). The type `PrimTypes` is a predefined set of primitive type names, e.g. {`Str`, `Bool`}. `Attrs` assigns to each class the set of its *attribute names*. Similarly `Assocs` assigns to each class the set of its *association names*, `AttrTys` assigns a primitive type to each attribute name of each class and `AssocTys` assigns a class to each association name which is to be thought of as the target class. Finally `multis` assigns a lower and an optional upper bound to each association, representing its multiplicity specification.



For example, the classes pictured above are defined by the ordering $\leq$ on `Cls` $= \{\mathsf{NE}, \mathsf{C}, \mathsf{A}\}$ where $\mathsf{NE} \leq \mathsf{C}$ and $\mathsf{NE} \leq \mathsf{A}$; `Attrs`(NE) $= \{\mathsf{n}\}$, `Attrs`(C) $= \{\mathsf{a}\}$, `Attrs`(A) $= \{\mathsf{m}\}$; `Assocs`(NE) $= \emptyset$, `Assocs`(C) $= \{\mathsf{attr}\}$, `Assocs`(A) $= \{\mathsf{o}\}$; the definition of `AttrTys` into `PrimTypes` is the obvious one; `AssocTys`(attr) $=$ A and `AssocTys`(owner) $=$ C. Similarly for multiplications by `multis`(attr) $=$ (0, nothing) and `multis`(owner) $=$ (1, just 1).

To interpret a meta-model, i.e. to provide the set of all its models, we first translate the above high-level specification into a more primitive form: a so-called dependent polynomial [3]. The so-called extension of a polynomial (I, A, B, s) – a *dependent polynomial functor* – is defined as an assignment of type $(I \to \mathrm{Set}) \to (I \to \mathrm{Set})$ defined as $\lambda\,(X : I \to \mathrm{Set}).\lambda\,(i : I).\, \Sigma_{a \in A_i} \Pi_{b \in B_{i,a}} X_{s\,b}$. Meta-models are interpreted as coalgebras for a dependent polynomial functor obtained from a translation of a meta-model specification into a dependent polynomial. In rough terms, an interpretation of a meta-model is a set of memory locations, P, together with an assignment, to each location $p \in P$, of a pair whose first component gathers all data related to the class stored at location p and its second component is a product of memory locations from P. See [5] for the details.

This setup naturally leads to the development of a modal logic for models of meta-model $\mathcal{M}$ where one would have, for all pointers p of type Class

$$\mathcal{M} \;\models_{\mathsf{Class}}\; \Box_{\mathsf{att}} \Diamond_{\mathsf{owner}} \mathsf{self} \;=\; \mathsf{self}$$

as the proposition that *some* owner of *all* p's attributes is p. In this fashion a meta model $\mathcal{M}$ defines a logic $\mathcal{L}(\mathcal{M})$ of constraints over its models. For the meta-model $\mathcal{M}_1 \otimes \mathcal{M}_2$, for a suitable notion $\otimes$ of tensor product of meta-models, $\mathcal{L}(\mathcal{M}_1 \otimes \mathcal{M}_2)$ allows one to express transformations by means of constraints binding all source and target models. A solution to such constraints yields a valid model-to-model transformation.

4 Automated theorem prover integration

When forming a model transformation by constraint satisfaction a large number of proofs are required. Agda provides a useful environment in which the various elements of a transformation can be constructed independently through meta-variable refinement. However, satisfying the many constraints is an enormous burden. It is therefore highly desirable to allow some proof automation.

An *automated theorem prover* (or ATP) is a program for solving problems in first-order logic. They are very fast, often providing solutions in a few seconds and are therefore very useful for quickly forming proof terms. When taken with the fact that within the domain of constructive logic proofs and programs are the same, it follows that an ATP can be used as an aid in semi-automated program construction. Nevertheless, it is necessary to first overcome the problem that most ATPs are *classical* and their proofs may be invalid in constructive logic.

As a first step we have integrated the *Waldmeister* [4] equational ATP into Agda. We therefore avoid the problem of Waldmeister being classical, because equational proofs do not use the principle of excluded middle. Furthermore, since many OCL constraints are equational this seems a reasonable compromise.

Our integration is achieved by means of a *reflection layer* which allows ATP problems and proofs to be manipulated in Agda. A proof goal is specified in Agda and, rather than proving it manually, the problem is reflected and serialised to Waldmeister input. Waldmeister is then executed (with a timeout), which will hopefully result in a proof which is parsed and converted back into the reflection layer by *proof reconstruction.* Proof reconstruction converts each step of the Waldmeister proof into an equivalent step at the reflection layer within Agda.

We have proven that each step of a reflection layer proof is sound with respect to Agda's native equality, and therefore the reflected proof can be realised as an Agda proof, completing the proof cycle. Further details can be found in [2].

As such our integration has only been applied to small examples, such as basic data-types like lists. Nevertheless, we have proven that such ATP integrations are both possible and useful. Future work will involve using our integration together with our meta-modelling environment to implement semi-automated derivations of model transformations.

5 Conclusion

We have briefly summarised our implementation of meta-models in Agda and the integration of an automated theorem prover to support the formal construction and anaysis of model transformations. Extending the meta-model with a full query logic as outlined will allow formal specification of model transformations as functions, with their construction aided by step-wise refinement and constraint satisfaction. Agda already provides a great number of tools to support program development, and we have further augmented these by a prototype ATP integration. This, we believe, makes Agda worthwhile considering as a tool for formally defining the logic of meta-models and object-oriented systems.

The next steps of this project consist in the implementation of model transformations in Agda, the improvement and extension of the theorem proving facilities, the development of refinement and optimisation techniques for transformations, and the development of case studies within our framework.

Agda promises a smooth approach for constructing transformations from declarative specifications while proving their correctness, but may not be the tool of choice for software engineers. It therefore needs to be hidden to a large extent as a backend for an Eclipse interface that provides the user with a simple declarative specification and development language. Our proof automation is a crucial ingredient for making that interface possible.

References

1. 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)
2. 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, pp. 116–130. Springer (2011)
3. Gambino, N., Hyland, M.: Wellfounded trees and dependent polynomial functors. In: Berardi, S., Coppo, M., Damiani, F. (eds.) Types for Proofs and Programs, LNCS, vol. 3085, pp. 210–225. Springer (2004)
4. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister: High performance equational deduction. Journal of Automated Reasoning 18(2), 265–270 (1997)
5. Poernomo, I., Rypacek, O.: A coalgebraic model of object systems stored on the heap (2011), under submission