Formal Refinement and Proof of a Small Java Program

Tony Clark

Department of Computing, University of Bradford, UK, BD7 1DP
a.n.clark@scm.brad.ac.uk

Abstract. The main components of a formal technique for specifying, refining and proving properties of object-oriented programs are presented. The technique is based on a λ-notation whose semantics is given using standard categorical constructs. An example of the formal development of a small Java program is presented.

1 Introduction

The aim of this work is to provide a rigorous framework for step-wise object-oriented software development which supports specification, refinement, proof and implementation. The framework takes the form of a categorical semantics of object-oriented system behaviour and a design language based on λ-notation.

This paper gives an overview of the main components of the framework using a simple system requirements and producing a Java program. It is not possible to give a full analysis of the approach in a paper of this length, the reader is directed to work by the author in the area of OO systems: [Cla96] [Cla94] [Cla97], [Cla98], [Cla99a], [Cla99b] and [Cla99c] and related work: [Ken99], [Ken97], [Eva98], [Eva99], [Bic97], [Lan98] and [Rui95] in formal methods for object-oriented development. The reader is directed to [Bar90], [Ehr91], [Gog75], [Gog89], [Gog90], [Pie96] and [Ryd88] for related work using category theory in systems development.

2 Development Framework

An *object state* is $<\alpha,\tau,\rho>$ where α is the object's type, τ is the object's identity and ρ is a partial function mapping attribute names to values. A *message* is $<\tau_s,\tau_t,\nu>$ where τ_s is the identity of the source object, τ_t is the identity of the target object and ν is a data value. Object-oriented system computation occurs in terms of state transitions resulting from message passing: $\ldots \longrightarrow \Sigma_1 \stackrel{(I,O)}{\longrightarrow} \Sigma_2 \longrightarrow \ldots$ in which a set of object states Σ_1 receives a set of input messages I producing a transition to states Σ_2 and output messages O. Since the behaviour of a system design may be non-deterministic it can be represented as a graph whose nodes are labelled with sets of object states and whose edges are labelled with pairs of sets of messages. This leads to a category **Obj** whose objects are graphs and whose arrows are graph homomorphisms.

System construction is described by standard categorical constructions in **Obj**. Given two behaviours O_1 and O_2 in **Obj** the product $O_1 \times O_2$ exhibits both O_1 and O_2 behaviour subject to structural consistency constraints. The co-product $O_1 + O_2$ exhibits either O_1 or O_2 behaviour. Equalizers, pull-backs and push-outs can be used to express constraints such that two or more behaviours are consistent. Computational category theory provides an algorithm for computing the behaviour of a system of inter-related components using limits.

Any behaviour O can be viewed as a category in which the objects are behaviour states and arrows are sequences of message pairs. Category-hood follows from: every object Σ has an identity arrow $[]$; and for every pair of arrows $f : \Sigma_1 \to \Sigma_2$ and $g : \Sigma_2 \to \Sigma_3$ there is an arrow $g \circ f : \Sigma_1 \to \Sigma_3$ which is constructed as $f +\!\!+ g$; and the associativity of $\circ$ follows from the associativity of $+\!\!+$. A refinement R is expressed as adjoint functors $R : O_1 \to O_2$ and $U : O_2 \to O_1$:

$$\begin{tikzcd} \Sigma_1 \arrow[r, "f"] \arrow[d] & U(\Sigma_2) \\ R(\Sigma_1) \arrow[r, "g"'] & \Sigma_2 \arrow[u] \end{tikzcd} \tag{1}$$

The diagram 1 states that performing a computation in the source object is the same as translating the source state, performing the computation in the target object and then translating the target state. Given any Σ_1 the refinement is *sound* if for every f there exists a g and is *complete* if for every g there is an f [Sab97].

Object-oriented designs are expressed using a λ-notation [Lan64] whose semantics is given by **Obj**. A behaviour is denoted by a functions M and is supplied with type, identity, attribute and message information: $M(\alpha)(\tau)(v)(I) = \bigcup_{i=1,n} \{(P_i, O_i)\}$ where P_i are *replacement behaviours* and O_i are corresponding output messages. This approach is essentially the same as that of Actor Theory [Agh86] [Agh91]. The basic model of message handling is *asynchronous*, however syntactic sugar can be used to express *synchronous* message passing. The following example shows how a behaviour function (left) which synchronously sends a message e_1 is translated to a behaviour function (right) which uses a replacement *wait*:

```
letrec agent(α)(τ)(σ)(m) =
   case m of
      p₁ →
         let p₂ ← e₁
         in e₂
   end
```

```
letrec agent(α)(τ)(σ)(m) =
   case m of
      p₁ → (agent(α)(τ)(σ) + wait, e₁)
         whererec wait(m) =
            case m of
               p₂ → e₂
               else (wait, ∅)
            end
   end
```

3 Development of a Java Program

The requirements for a library system are defined. An initial object-oriented design is constructed. A single refinement step is performed and verified. A simple system property is established. The design is analysed prior to translating it to an implementation in Java (appendix A).

Software to control a library is required. The library has readers who may borrow copies of books. At any given time each reader has a number of books on loan. New readers may join the library at any time. The library has a number of copies of books. Each book has a unique title. A copy is either on the shelf in the library or is being borrowed by a reader. Libraries operate a shares readership policy whereby joining one library permits readers to borrow books at all participating libraries.

A library system consists of a single object with a state (R, B) consisting of readers R and books B. Each reader is a pair (n, C) where n is a name and C is a set of borrowed copies. Each book is a pair (n, i) where n is a name and i is the number of shelved copies. Initially we treat R and B as lookup tables. Let T be a table with keys $dom(T)$, lookup is $T \bullet k$, extension is $T[k \mapsto v]$. Adding table values is defined as follows (removing is similarly defined):

$$T[k \oplus v] \equiv \begin{cases} T[k \mapsto T \bullet k \cup \{v\}] & \text{when } \mathit{isSet}(T \bullet k) \\ T[k \mapsto T \bullet k + v] & \text{when } \mathit{isInt}(T \bullet k) \end{cases}$$

Initial system behaviour can be decomposed into the success and failure modes. The design operator + allows us to define these modes separately and then combine them. Success mode is defined as follows:

$$
\begin{array}{l}
\textbf{letrec } \mathit{libOk}(\alpha)(\tau)(R,B)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\qquad \mathit{addReader}(n) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R[n \mapsto \emptyset], B), \emptyset) \textbf{ when } n \notin \mathit{dom}(R) \\
\qquad \mathit{addBook}(n) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R, B[n \mapsto 0]), \emptyset) \textbf{ when } n \notin \mathit{dom}(B) \\
\qquad \mathit{addCopy}(n) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R, B[n \oplus 1]), \emptyset) \textbf{ when } n \in \mathit{dom}(B) \\
\qquad \mathit{borrow}(n_1, n_2) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R[n_1 \oplus n_2], B[n_2 \ominus 1]), \emptyset) \\
\qquad\qquad\qquad\qquad\qquad \textbf{when } n_1 \in \mathit{dom}(R) \,\&\, n_2 \in \mathit{dom}(B) \\
\qquad \mathit{return}(n_1, n_2) \rightarrow \mathit{libOk}(\alpha)(\tau)(R[n_1 \ominus n_2], B[n_2 \oplus 1]), \emptyset) \\
\qquad\qquad\qquad\qquad\qquad \textbf{when } n_1 \in \mathit{dom}(R) \,\&\, n_2 \in \mathit{dom}(B) \\
\qquad \textbf{else } (\mathit{libOk}(\alpha)(\tau)(R, B), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

Given a state (R, B) in the source behaviour, a refinement acts as identity on R and transforms $B = \{n_1 \mapsto i_1, \ldots, n_k \mapsto i_k\}$ into a set of object identifiers $\{\tau_1, \ldots, \tau_k\}$ and introduces new objects $\tau_1 \mapsto (n_1, i_1), \ldots, \tau_k \mapsto (n_k, i_k)$ to the system state. A book behaviour is as follows:

$$
\begin{array}{l}
\textbf{letrec } \mathit{book}(\alpha)(\tau)(n,i)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\qquad <\tau', \tau, \mathit{getName}> \rightarrow (\mathit{book}(\alpha)(\tau)(n,i), \{<\tau, \tau', n>\}) \\
\qquad \mathit{borrow} \rightarrow (\mathit{book}(\alpha)(\tau)(n, i-1), \emptyset) \textbf{ when } i > 0 \\
\qquad \mathit{addCopy} \rightarrow (\mathit{book}(\alpha)(\tau)(n, i+1), \emptyset) \\
\qquad \textbf{else } (\mathit{book}(\alpha)(\tau)(n,i), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

The successful library behaviour is modified to take account of book objects. The initial design uses set membership to test for the existence of a book. This must now be implemented as a private method of the library:

$$
\begin{array}{l}
<\tau', \tau, \mathit{findBook}(\emptyset, n)> \rightarrow (\mathit{libOk}(\alpha)(\tau)(R,B), \{<\tau, \tau', \mathit{noBook}>\}) \\
<\tau', \tau, \mathit{findBook}(\{o\} \cup S, n_1)> \rightarrow \\
\quad \textbf{let } n_2 \leftarrow <\tau, o, \mathit{getName}> \\
\quad \textbf{in if } n_1 = n_2 \\
\qquad \textbf{then } (\mathit{libOk}(\alpha)(\tau)(R,B), \{<\tau, \tau', \mathit{book}(o)>\}) \\
\qquad \textbf{else } (\mathit{libOk}(\alpha)(\tau)(R,B), \{<\tau', \tau, \mathit{findBook}(S, n_1)>\})
\end{array}
$$

When a library receives an *addBook* message with a name n which does not already exist then a new book object is created. We assume that τ'' is a new object identifier and that β is the type tag for books:

$$
\begin{array}{l}
\mathit{addBook}(n) \rightarrow \\
\quad \textbf{let } \mathit{noBook} \leftarrow <\tau, \tau, \mathit{findBook}(n)> \\
\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R, B \cup \{\tau''\}) \times \mathit{book}(\beta)(\tau'')(n, \emptyset), \emptyset)
\end{array}
$$

To verify the refinement step the following source state is used: $\{\tau \mapsto (R,B)\}$ where R is a set of readers and B is the set $\{n_1 \mapsto i_1, \ldots, n_k \mapsto i_k\}$. The corresponding target state is $\{\tau \mapsto (R,T)\} \cup O$ where T is the set of object identifiers $\{\tau_1, \ldots, \tau_k\}$ and O is the state $\{\tau_1 \mapsto (n_1, i_1), \ldots, \tau_k \mapsto (n_k, i_k)\}$. The refinement of *addBook* is sound and complete when the following diagram commutes (see diagram 1):

$$
\begin{tikzcd}
\{\tau \mapsto (R,B)\} \arrow[r, "\mathit{addBook}(n)"] \arrow[d] & \{\tau \mapsto (R, B[n \mapsto 0])\} \\
\{\tau \mapsto (R,T)\} \cup O \arrow[r, "c \circ \mathit{addBook}(n)"'] & \begin{array}{c}\{\tau \mapsto (R, T \cup \tau'')\} \cup \\ O[\tau'' \mapsto (n,0)]\end{array} \arrow[u]
\end{tikzcd} \qquad (2)
$$

A proof of 2 is by induction on the size of the set B and the length of the computation c. Further refinement identifies a class of behaviours for *reader* and adds a private method *findReader* to the library.

The design language is given a formal semantics in terms of standard constructions in **Obj**. A design language proof theory provides a framework for establishing program properties. The proof theory views a behaviour function as a mapping from input messages and states to output messages and states. Proofs typically are by induction on the length of a messages stream. Since refinement is formally defined, it is possible to show that properties are preserved by refinement transformations.

Consider the following theorem. For any library (R,B), if b is a book borrowed by a reader then $b \in \mathit{dom}(B)$. The proof is by induction in the length of the input message stream. The theorem holds for library $(\emptyset, \emptyset)$ and the empty stream. Assume by induction that the theorem holds for library (R, B) and messages ms. Now show by case analysis on m that the theorem holds for all messages $ms +\!\!+ [m]$. We conclude that the theorem holds.

Consider the behaviours *book* and *reader*. Both provide a state component n which is used to index into collections of behavioural instances using the message *getName*. This indicates that there is a common behaviour *named* and projection morphisms. In an implementation *named* will occur as a super-class of both *book* and *reader*.

$$\begin{tikzcd} & \mathit{named} & \\ \mathit{book} \arrow[ur] & & \mathit{reader} \arrow[ul] \end{tikzcd}$$

Consider a behaviour functor F_1 which acts on system states by projecting all book objects to equivalent named objects by forgetting the copy count. F_1 acts as identity on all arrows except that $\mathit{findBook}(O, n)$ is replaced by $\mathit{find}(O, n)$, $\mathit{book}(b)$ is replaced by $\mathit{found}(b)$ and *noBook* is replaced by *notFound*.

In order for F_1 to be valid, it must be sound and complete with respect to indexing into collections of books. Therefore, for any system state Σ, the diagram on the right must commute. Similarly, a behaviour functor F_2 is defined to project states and calculations involving indexing readers. This leads us to replace the behaviours for *findBook* and *findReader* with a single behaviour *find*.

$$\begin{tikzcd} \Sigma \arrow[r, "{[\mathit{findBook}(O,n)] +\!\!+ c}"] \arrow[d] & \Sigma \arrow[d] \\ F_1(\Sigma) \arrow[r, "{[\mathit{find}(O,n)] +\!\!+ F_1(c)}"'] & F_1(\Sigma) \end{tikzcd}$$

The shared readership policy is expressed as a pull-back S on a diagram showing two (or more) libraries which project onto a behaviour *cell* containing their readers. The pull-back ensures that both libraries have the same readers. There are a number of implementation choices for the shared readership policy whose behaviour is defined by S. If the programming language supports shared data between class instances (such as `static` in Java) then the R component of a library class may be shared.

$$\begin{tikzcd} & S \arrow[dl] \arrow[dr] & \\ \mathit{lib}_1 \arrow[dr, "\gamma_1"'] & & \mathit{lib}_2 \arrow[dl, "\gamma_2"] \\ & \mathit{cell} & \end{tikzcd}$$

References

[Agh86] Agha, G.: Actors: *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Agh91] Agha, G.: The Structure and Semantics of Actor Languages. In proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, LNCS 489, Springer-Verlag, 1991.

[Bar90] Barr, M. & Wells, C.: *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science, 1990.

[Bic97] Bicarregui, J., Lano, K. & Maibaum, T.: Towards a Compositional Interpretation of Object Diagrams. Technical Report, Department of Computing, Imperial College, 1997.

[Cla94] Clark, A. N.: A Layered Object-Oriented Programming Language. GEC Journal of Research, 11(3), The General Electric Company p.l.c., pp 173 – 180, 1994.

[Cla96] Clark, A. N.: *Semantic Primitives for Object-Oriented Programming Languages*. PhD Thesis, QMW, University of London, 1996.

[Cla97] Clark, A. N. & Evans, A. S.: Semantic Foundations of the Unified Modelling Language. In the proceedings of the First Workshop on Rigorous Object-Oriented Methods: ROOM 1, Imperial College, June, 1997.

[Cla98] Clark, A. N.: Type Checking OCL Expressions. Technical Report, 1998.

[Cla99a] Clark, A. N.: A Semantics for Object-Oriented Systems. Presented at the Third Northern Formal Methods Workshop. September 1998. To appear in BCS FACS Electronic Workshops in Computing, 1999.

[Cla99b] Clark, A. N.: A Semantics for Object-Oriented Design Notations. Technical report, submitted to the BCS FACS Journal, 1999.

[Cla99c] Clark, A. N.: A Semantic Framework for Object-Oriented Development. Technical report, submitted to the L'Objet journal special issue on formal object-oriented development, 1999.

[Ehr91] Ehrich, H-D., Goguen, J. A. & Sernadas, A.: A Categorical Model of Objects as Observed Processes. In the proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, LNCS 489, Springer-Verlag, 1991.

[Eva98] Evans, A. S.: Reasoning with UML Class Diagrams. In WIFT '98, IEEE Press, 1998.

[Eva99] Evans, A. S. & Lano, K. C.: Rigorous Development in UML. To appear in the proceedings of the ETAPS '99, FASE Workshop, 1999.

[Gog75] Goguen, J.: Objects. Int. Journal of General Systems, 1(4):237–243, 1975.

[Gog89] Goguen, J.: A Categorical Manifesto. Technical Report PRG-72, Programming Research Group, Oxford University, March 1989.

[Gog90] Goguen, J. A.: Sheaf Semantics for Concurrent Interacting Objects. Mathematical Structures in Computer Science, 1990.

[Ken99] Kent, S. & Gil J.: Visualising Action Contracts in Object-Oriented Modelling. To appear in the IEE Software Journal, 1999.

[Ken97] Kent, S.: Constraint Diagrams: Visualising Invariants in Object-Oriented Models. In the proceedings of OOPSLA 97, ACM Press, 1997.

[Lan64] Landin P.: The Next 700 Programming Languages. Communication of the ACM, 9(3), 1966, pp 157 – 166.

[Lan98] Lano, K. & Bicarregui, J.: UML Refinement and Abstraction Transformations. In the proceedings of the Second Workshop on Rigorous Object-Oriented Methods: ROOM 2, Bradford, May, 1998.

[Pie96] Piessens F. & Steegmans E.: Categorical Semantics for Object-Oriented Data Specifications. In *Formal Methods and Object Technology*, (eds.) Goldsack, S. J. & Kent, S. J., Springer-Verlag, 1996, pp 302 – 316.

[Rui95] Ruiz-Delgado, A., Pitt, D. & Smythe, C.: A Review of Object-Oriented Approaches in Formal Specification. The Computer Journal, 38(10), 1995.

[Ryd88] Rydeheard, D. E. & Burstall, R. M.: *Computational Category Theory*. Prentice Hall International Series in Computer Science, 1988.

[Sab97] Sabry, A. & Wadler, P.: A Reflection on Call-by-Value. ACM Transactions on Programming Languages and Systems, 19(5), pp 111 – 136,1997.

A Library Implementation in Java

Each independent behaviour is defined as a Java class. The state components of the behaviour are defined as fields and the message handlers are defined as methods. Any common behaviour is defined using inheritance. The main features are: the class `Named` defines the common behaviour for readers and books; attribute `readers` in `Library` is declared `static` so that libraries implement the shared readership policy; class `Library` defines a method `find` that is used to index both readers and books.

```
class Named {
  private String name;
  public Named(String name) { this.name = name; }
  public String getName() { return name; }
}

class Book extends Named {
  private int copies = 0;
  public Book(String name) { super(name); }
  public void borrow()
  {
    if(copies > 0)
      copies = copies - 1;
    else throw new Error("no copies left");
  }
  public void addCopy() { copies = copies + 1; }
}

class Reader extends Named {
  private Vector copies = new Vector();
  public Reader(String name,Vector copies)
  {
    super(name);
    this.copies = copies;
  }
  public void borrow(String name) { copies.addElement(name); }
  public void ret(String name) { copies.removeElement(name); }
}

class Library {
  private static Vector readers = new Vector();
  private Vector books = new Vector();
  public void addReader(String name) { readers.addElement(new Reader(name,new Vector())); }
  public void addBook(String name) { books.addElement(new Book(name)); }
  public void addCopy(String bookName)
  {
    Book book = (Book)find(bookName,books);
    if(book != null)
      book.addCopy();
    else throw new Error("cannot find book");
  }
  private Named find(String name,Vector table)
  {
    Named named = null;
    for(int i = 0; (named == null) && (i < table.size()); i++) {
      Named n = (Named)table.elementAt(i);
      if(n.getName().equals(name))
        named = n;
    }
    return named;
  }
  public void borrow(String readerName,String bookName)
  {
    Reader reader = (Reader)find(readerName,readers);
    Book book = (Book)find(bookName,books);
    if((reader != null) & (book != null)) {
      reader.borrow(bookName);
      book.borrow();
    } else throw new Error("illegal name in borrow");
  }
  public void ret(String readerName,String bookName)
  {
    Reader reader = (Reader)find(readerName,readers);
    Book book = (Book)find(bookName,books);
    if((reader != null) & (book != null)) {
      reader.ret(bookName);
      book.addCopy();
    } else throw new Error("illegal name in ret");
  }
}
```