Automated Method Induction

Functional Goes Object Oriented

Thomas Hieber, Martin Hofmann

University Bamberg - Cognitive Systems Group

thomas-wolfgang.hieber@stud.uni-bamberg.de, martin.hofmann@uni-bamberg.de

Abstract

The development of software engineering has had a great deal of benefits for the development of software. Along with it came a whole new paradigm of the way software is designed and implemented - object orientation. Today it is a standard to have UML diagrams automatically translated into program code wherever possible. However, as few tools really go beyond this we demonstrate a simple functional representation for objects, methods and variables. In addition we show how our inductive programming system *Igor* can not only understand those basic notions like referencing methods within objects or using a simple protocol like something we called *message-passing*, but how it can even learn them by a given specification - which is the major feature of this paper.

Keywords Inductive Programming, Object Oriented Programming, Igor, Maude, Java, Recursion

1. Introduction

Since mainstream software for business use is commonly not created with functional programming languages it is about time to raise the question whether it is possible to adapt object oriented language features to a functional setting. *Igor* is a system for synthesizing recursive functional programs, which learns recursive functions solely from input/output (I/O) examples, and will henceforth be our system of choice concerning program induction. Since *Igor* is naturally based on functional programming, the main focus of this paper lies on finding a way to use *Igor* for program inference in an object oriented background. This requires us to express the behavior of objects and method calls by I/O examples. In order to do so, it is necessary to find a way to express object oriented programs in a functional way. The main part of this paper will be concerned with this task on a very general scale. It is not meant to be a complete approach but an analysis of what is possible and what is not.

At the same time, it is necessary to enable an object oriented programmer to provide input to the synthesis system as unobtrusive as possible. For this purpose, we have devised an interface for Eclipse which will allow a programmer to use annotations in order to provide input for our induction process, thus seamlessly integrating with software engineering tools like *Rational Software Architect (RSA)*. Since it is not the focus of this paper to explain the functionality of this prototype we only mention it for the sake of completeness - more on this subject is to be found in [Hieber 2008].

In section 2 we start off with a short survey of current and past research in the field of functional programming and object orientation. Section 3 is a short roundup about *Inductive Programming (IP)* and *Igor* while section 4 focuses on how we represent object orientation to *Igor* and how it can even learn all of the concepts created. The prototype plug in for *Eclipse*, which is a further result of our research, will then be described very shortly in section 5 before we finally conclude in section 6.

2. The Status Quo

In the past 30 years, many different inductive programming (IP) systems have been developed, many of them sharing a functional approach. The extraction of programs from input/output examples started in the the seventies and has been greatly influenced by Summers' [Summers 1977] paper on the induction of *LISP* programs. After the great success of *Inductive Logical Programming (ILP)* on classification learning in the nineties, the research focus shifted more to this area. Prominent ILP systems for IP are for example *FOIL* [Quinlan and Cameron-Jones 1993], *GOLEM* [Muggleton and Feng 1990] or *PROGOL* [Muggleton 1995] - systems which make use of *Prolog* and predicate logic.

Later, the functional approach was taken up again by the analytical approaches *Igor1* [Kitzelmann and Schmid 2006], and *Igor2*[Kitzelmann 2007] and by the evolutionary/generate-and-test based approaches *Adate*[Olsson 1995] and *MagicHaskeller*[Katayama 2007].

All in all you can subsume the concern of *Inductive Programming* as the search for algorithms which use as little additional information as possible to generate correct computer programs from a given minimal specification consisting of input/output examples.

At the same time functional languages have had to face the development in programming paradigms which led to many approaches to support object orientation. Established functional languages have their own object oriented extensions like OCaml [Remy and Vouillon 1998] or OOHaskell ([Kiselyov and Laemmel 2005]). Additionally there are various approaches to include an object system into a functional language without changing the type system or the compiler (see e.g. [Kiselyov and Laemmel 2005] for a Haskell related overview).

For our purpose we do not need such sophisticated techniques (yet), therefore we content ourselves with taking on a quite naive and very simplified perspective, though sufficient for our case, and treat objects merely as tuples.

On the other hand there are some very powerful tools for object oriented programmers which support automated code-generation to a certain extent and the community for *Automated Software Engineering* is very productive to take this even further. In this context it is inevitable to have a look at program synthesis since we ideally do not want to stop at automatically generating class files from UML diagrams like IBM's *RSA*, or generate a GUI by 'WYSIWYG' editors such as *NetBeans* or *Visual Studio*.

3. Inductive Programming & IGOR

Summers' theories have been taken up again in [Schmid 2003], where the *Igor* system was put into existence. The basic idea is to generate a set of (recursive) equations from a specification consisting of input/output examples. Its first system was written in *LISP* and it was closely connected to Summers' suggestions. A few years later a newer version of *Igor* was created and it extended the prior version by a number of improvements. In [Kitzelmann 2008] you find a more detailed description of *Igor2* as a system which now employed mechanics such as anti-unification of the initial input/output examples and a *best-first* search over succeeding sets of equations which are to be formed by *term-rewriting*. Since this shift in the way programs were now processed did not play to the strengths of *LISP* like the former version, *Igor2* was written in the reflective term-rewriting language *Maude*.[1]

In order to understand how the system works before we go ahead and use it, let us consider the following example of the list-operator *length*.

Listing 1: Input/Output Examples for Igor

```
length([]) = 0
length([y]) = succ(0)
length([y,x]) = succ(succ(0))
length([y,x,z]) = succ(succ(succ(0)))
```

Given those examples, *Igor* correctly identifies the following recursive program:

Listing 2: Recursive Program for length

```
sub1(cons(x0,x1)) = length(sub2(cons(x0,x1)))
sub2(cons(x0,x1)) = x1
length[] = 0
length(cons(x0,x1)) = succ(sub1(cons(x0,x1)))
```

[1] see http://maude.cs.uiuc.edu/

This is what *Igor* produces from our specification (we have only adjusted the syntax to increase readability), especially the two functions **sub1** and **sub2** have been automatically introduced by the system. The *succ* operator is the well known *successor* in order to define the natural numbers as *peano numbers*. Evidently this system's output is purely functional so we are going to be concerned with finding out how to bring this closer to an object oriented context.

4. Igor and Object Orientation

As already mentioned, *Igor* is firmly based within the functional paradigm along with all its strengths and weaknesses. Nevertheless it is going to be subject of our concern in which way it could be possible to represent an object-oriented specification with *Maude* and feed it to *Igor*. For this we are going to put together some example specifications, have them synthesized and evaluate the output. In order to do so it is important to understand how we could possibly map the way object-oriented programs are presented to a functional notation. We are going to deal with this problem's theory first and then try and find out how *Igor* will react to our input.

When we are dealing with *Maude* specifications in the following chapters, let the following notation be established:

$[object].[datatype]$

This is the way data types are represented by *Maude* and we will stick to it for the sake of transparency. For the *Maude* results we will also establish a notation since the code generated is not quite readable. So the way results will be displayed like this:

$ResultRule/Pattern(EquationLHS) = (EquationRHS)$

4.1 Representing the Object

In this attempt we will try and keep it simple, as we are only exploring so the motto is to start small. When thinking about objects we can agree that they basically consist of an **identifier**, a set of **(member) variables** [2] and a set of **methods**. So it seems quite advisable to represent any object like this:

$identifier.String \times variables.List \times methods.List \rightarrow Object$

Let us for now just take variables and methods as black boxes, we will deal with them after this. Apart from the elaboration on those, there are only two things left in order to get a basic quasi object-oriented system: **calls** and **exceptions**. The former is the basic notion of a messages sent between objects in our system. The latter are a vital part of any high level programming language and, more importantly, we are going to need them in order to correctly specify some of our components. Since error handling is not the major part of our

[2] a.k.a. properties - see section 4.4

concern, let it just be introduced as black box - we are not going to analyse it any further.

Messages shall be defined like this:

$$ParamList \times Object \rightarrow Message$$

So a message consists of a number of arguments (*ParamList*) and an object which in case of a function call can carry the return value back to the sender, which leaves us with the following definition of the object in the *Maude* specification:

Listing 3: Object Constructor

```
op ___ : Identifier VarList MethodList -> Object [ctor] .
```

Note the ___ as the constructor's name - it is *Maude* syntax for n-ary operators with blanks as constructors (three underscores → three parameters). The constructor uses an *identifier*, a set of *variables* and a list of *methods* in order to create a new object. Now that we have an idea of how to represent an object, let us try and find out how we can do the same thing on methods.

4.2 Representing the Method

Before going on we have to bear in mind that - for now - we are dealing with methods only on a syntactic level. We only want to find out how to represent them in the context of an object. We are not concerned with the procedures within the method's body nor with how they are used. All we need to know for now is what information we need about a method on the object level in order to keep it as abstract as possible. Remember that we want to have this representation to be kept within the *MethodList* in our newly defined object. Right now we can say that a representation of a method must contain the following information:

- Method Name
- Return Value
- Argument Specification

When we formally put this together it ends up looking like this:

$$identifier.String \times return\_value.DType \times arguments.List \rightarrow Method$$

The *DType* is again to be taken as black box here since we are not interested in type inference or casting, so to understand that it is necessary information for any object calling the method is enough in this context.

This leaves us with the definition in *Maude* as follows:

Listing 4: Method Constructor

```
op met : Identifier DType ParamList -> Method [ctor].
```

As we have seen before, this is basic Maude notation for defining an operator called *met*. It takes three parameters (Identifier, DType and ParamList) and produces a *Method*. Since this is a classical constructor the *[ctor]* command is used at the end of the definition.

4.3 Representing the Method Call

Before we can actually call a method we have to resolve the identifier within the object which supposedly encapsulates it. In order not to become too confusing we are going to step away from objects for one moment and just focus on the way you might find a method within an object. For this let us assume that there exists a method list as depicted in 4.1 and an object trying to call a method by an identifier. The idea is to get a matching process like:

$$Identifier \times MethodList \rightarrow Method$$

By now we have introduced a few basic notations in object orientation. They all share the tuple-structure which is important in order to build the bridge to functional programming. As a consequence, these concepts can now be modelled in *Maude* (see section A.1) making it easy for us to construct simple examples in order to demonstrate how *Igor* responds to them. As a start we have picked the 'identifier-match' which is the mapping procedure we have just introduced. The full *Maude* specification is to be found in listing16 in this section we will only display short snippets in order to illustrate.

Listing 5: Identifier Match

```
sorts List Method Identifier DType ParamList NPException
    .
 subsort Method < NPException .
```

In the first part there are some *sort* definitions which are quite obvious and should be familiar by now. The only slightly strange thing is the second line. Here we basically bring in the exception since we want a *NPException (= Null Pointer Exception)* to be thrown in case an identifier is not found within the method list. The exception is here derived from *Method* which is obviously not very elegant or - strictly spoken - even wrong. But since we have not yet constructed a well defined object framework we can forsake the strict rules which would come along with it and just have the exception be the subclass of *Method*. This gives us the chance to explain another concept in Maude - sorts and subsorts. You can see that there is a number of sorts defined, was well as a relation between *Method* and *NPException*. The operator used here is < which can be seen like an arrow pointing from the specific to the more general sort.

The next part of the specification (listing 6) gives us some constructors and variables before we can go ahead and define our input examples.

Listing 6: Identifier Match - Constructors and Variables

```
op [] : -> List [ctor] .
op cons : Method List -> List [ctor] .
op mm : Identifier DType ParamList -> Method [ctor] .
ops id1 id2 id3 : -> Identifier .
op parlist : -> ParamList [ctor] .
op exc : -> NPException .
op dt : -> DType .
op match : Identifier List -> Method [metadata "induce"]
    .

vars m1 m2 m3 : Method .
```

Here we find a basic procedure of constructing a list (ll 1,2), a method (*mm* operator), some random identifiers, arguments, an exception as well as a datatype. Note that identifiers, arguments, exception and datatype are just instantiated without any concrete data attached but for the current level of abstraction it is not necessary to do so. The operator *match* now is the method to be induced by *Igor* and after declaring a few variables as methods all there is left to do is to assert our input/output examples.

Listing 7: Identifier Match - Input/Output Examples

```
eq match(id1, [] ) = exc .

eq match(id2, [] ) = exc .

eq match(id1, cons(mm(id1, dt, parlist) ,[]) ) =
 mm(id1, dt, parlist) .

eq match(id1, cons(mm(id2, dt, parlist), []) ) = exc .

eq match(id2, cons(mm(id1, dt, parlist) ,[]) ) = exc .

eq match(id2, cons(mm(id2, dt, parlist), []) ) =
 mm(id2, dt, parlist) .

[...]
```

The equations in listing 7 are used to give *Igor* some basic examples in the problem domain. Here we bring together what we have defined earlier (Listings 5 and 6). The first two are quite obvious and finally explain why we insisted on exceptions earlier. Of course there could just be an empty method as a return value, but since we are trying to conquer the object oriented world with *Igor*, it feels more natural to express it this way. All the other examples (see complete listing 16) are summarised quite quickly - every time the method called is contained in the method list it is returned.

If this is now fed to *Igor*, one of the resulting hypotheses (translated into a little more readable syntax) returned is a set of equations. $X1$ and $X2$ are identifiers, $X3$ is a list, *dt* a datatype and $parlist$ a list of parameters:

1. $match(X1, []) = exc$
2. $Sub1(X1, cons(mm(X2, dt, parlist), X3)) =$ $case\,(X1 == X2)\,of\,False\, \rightarrow X1$
3. $Sub2(X1, cons(mm(X2, dt, parlist), X3)) =$ $case\,(X1 == X2)\,of\,True\, \rightarrow X3$
4. $match(X1, cons(mm(X2, dt, parlist), X3)) =$ $case\,(X1 == X2)\,of\,False\, \rightarrow$ $match(Sub1(X1, cons(mm(X2, dt, parlist),$ $case\,(X1 == X2)\,of\,True\, \rightarrow$ $mm(X1, dt, parlist)$

From this simple example we can already see how *Igor* tackles this problem. The *base case* is the first equation. Equations 2 to 4 ensure that the number of methods in the list is gradually decreased every time the first method in the list does not correspond to the one called. So at the end the list of methods becomes either void (→ equation 1) or the method is found at the head of the current method list (→ equation 5).

Already we can observe how *Igor* tries to find a recursive solution to this problem, which may seem a little complicated for this purpose, but it is exactly what we wanted to achieve and so we can go on at this point knowing that *Maude* and *Igor* can handle what we outlined earlier.

4.4 Concerning Variables

Before going on it is necessary to point out that when we talk about variables in this paper we mean actually member variables (properties), so from now on these two concepts will be used synonymously. For our purpose, variables are very similar to methods. They just happen to be much more simple since there is no need for a list of arguments to be carried around. This comes all down to this simple line in our object specification in *Maude*:

Listing 8: Variable Constructor

```
op var : Identifier DType -> Variable [ctor].
```

The way a variable is referenced is exactly the same as we have just done it with methods just that our *MethodList* would now be a *VariableList* - so there is no point in repeating the procedure all over.

4.5 Messages

As already mentioned, we are going to relate every action within our system to messages. In 4.1 we have defined the specification of them and this is how they look in *Maude*:

Listing 9: Message Constructor

```
op msg : ParamList Object -> Message [ctor] .
```

We have seen how the matching of identifiers works, so let us now find out about messages sent between two imaginary objects. Since we are now only concerned with the way data is wrapped within them we drop overhead like identifiers and the like for now and focus on the core procedure which takes a message and its arguments and returns an object as result value.

We are going to test this with an example problem - the *even* operation which determines if a number is even or not. As before, we first have to define a couple of sorts.

Listing 10: Identifier Match Sorts

```
sorts InVec Object .
sorts Message ParamList .
sorts Nat Bool Param .
 subsorts Param < Nat Bool .
 subsorts Object < Nat Bool .
```

As we want to compute some real data this time, we have to refer *Param* and *Object* to real values as we do here.

Listing 11: Identifier Match Constructors

```
op <> : -> ParamList [ctor] .
op msg : ParamList Object -> Message [ctor] .
op null : -> Object [ctor] .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op t : -> Bool [ctor] .
op f : -> Bool [ctor] .
op cpar : Param ParamList -> ParamList [ctor] .

op method : Message -> Message [metadata "induce"] .
```

Next to the already known definitions of *message* and the usual list operations there are some more definitions. In addition to the *successor* operator (we call it just *s* this time) we need the boolean values *true (t)* and *false (f)*. What we want for *Igor* to do now is to unwrap a message, take the argument list as input and put the result back into a message. Formulated with input/output examples this is what we get:

Listing 12: Identifier Match Input/Output Examples

```
eq method( msg(cpar( 0,  <>), null) ) =
 msg(<> ,t ) .

eq method( msg(cpar( s(0),  <>), null) ) =
 msg( <> , f ) .

eq method( msg(cpar( s(s(0)), <>), null) ) =
  msg( <> , t ) .

eq method( msg(cpar( s(s(s(0))), <>), null) ) =
 msg( <> , f ) .

eq method( msg(cpar( s(s(s(s(0)))), <>), null) ) =
 msg( <> , t ) .
```

So we assume that *Igor* simulates an object getting a message with a natural number as parameter, returning a message containing a boolean. Now we will once again run this through the system and get the following set of equations (with $X1$ being a natural number):

1. $Sub19(msg(cpar(s(s(X1)), <>), null)) = msg(cpar(X1, <>), null)$
2. $method(msg(cpar(0, <>), null)) = msg(<>, t)$
3. $method(msg(cpar(s(0), <>), null)) = msg(<>, f)$
4. $method(msg(cpar(s(s(X1)), <>), null)) =$
   $method(Sub19(msg(cpar(s(s(X1)), <>), null)))$

On the level of semantics this looks just like what we wanted. On every left hand side there is a message with arguments and the right hand side contains messages with return value. So *Igor* has learnt the concept of message-passing, but since we provided a real problem specification encapsulated within the message this time, we will have to evaluate the resulting program for functional validity also. For this it seems appropriate to take off the wrapping from the synthesised equations and just show the important bits.

1. $Sub19(s(s(X1))) = X1$
2. $method(0) = t$
3. $method(s(0)) = f$
4. $method(s(s(X1))) = method(Sub19(s(s(X1))))$

Now this looks just like what we intended. Equations 2 and 3 are the *base cases*, 4 and 1 make sure that any number bigger than 1 will gradually be reduced by two until one of the *base-cases* is reached. Then the result value is ultimately returned.

4.6 Back Into Perspective

Before we try and draw a conclusion out of the results learned let us quickly summarize what we have gained so far. We have modelled a simple object-oriented protocol consisting of **Objects**, **Variables**, **Methods** and **Message-Calls**. Modelling those concepts in a functional way has brought our program synthesis system - *Igor* - to understand and even 'learn' simple routines like 'identifier-matching' and 'message-passing'. Now for a final test let us have a look how it can handle some of the syntactic sugar which is widespread in object oriented programming languages - a simple iteration over a collection. The task is to take a set of abstract objects and apply a method to every object within the set (which is actually a list). In listing 13 (proper Maude example in listing 20) we take one collection of objects and as we iterate over them we apply a method to them and put the results into a new collection. The results are represented in listing 21.

Listing 13: Iterate Collection Input/Output Examples

```
eq iterate([]) = {} .
eq iterate( put(Y,[]) ) = put2( met(Y), {}) .
eq iterate( put(X,put(Y,[])) ) = put2( met(X), put2( met(
    Y) ,{})) .
[...]
```

As you can see from the equations in listing 13 we employ two different collections and along with it two different constructors *put* and *put2*, which are like a *cons* operator. This is not necessary but in order to illustrate that we are actually removing the objects from one to another collection it seems to be more appropriate.

In our second example in listing 22 we go the same way we already did with methods and objects. We expand our simple iteration example by enhancing the method call itself. Another layer of abstraction is added or, if you will, some more object oriented 'overhead' by adding more detail into the method call like identifier and the parent object the method is to be invoked on. Now it is not just *met(Y)* but a method call specified like this:

$$object.Object \times identifier.String \times return\_value.DType \times arguments.ParamList \rightarrow Method$$

The result (listing 21) shows that, like before, all the additional information is just wrapped around the detected procedure which still does nothing else than moving objects from one collection to another. The following equations display the result in a more readable notation. Note that $X1$ is an *Object*, $X2$ a *Collection*, $id1$ an *Identifier*, $dt1$ a *DType* and *pp* a *ParamList*.

1. $Sub1(push(X1, X2)) = call(X1, id1, dt1, pp)$
2. $Sub2(push(X1, X2)) = it\_apply(Sub5(push(X1, X2))))$
3. $Sub5(push(X1, X2)) = X2$
4. $it\_apply([]) = []$
5. $it\_apply(push(X1, X2)) =$
   $push(Sub1(push(X1, X2)), Sub2(push(X1, X2)))$

Not only have we modeled primitive object-oriented concepts - we have had *Igor* synthesize them on its own just by providing some generic input/output examples. After that we went one step further trying to model some object-oriented procedures like iteration or the 'foreach' loop just by using those primitives. It turned out that *Igor* does not appear to be struggling with the example specification - even though we have wrapped them in quite a complex model - especially in the 'foreach' example. We have constructed some more examples to show that we can now use our primitive objects to build a more complex model consisting solely of the concepts illustrated in this section. This means that it is possible to take this further, modeling a complete object-oriented model just with a functional programming language. At the same time it has to be said that we did indeed skip quite a lot of things as type-inference, inheritance, references (pointers) or exceptions, to name but a few. Since it has been pointed out that the model constructed in this paper does not claim to be a full scale approach we cannot conclude that a serious foundation has been created to build on.

But we have demonstrated that it is basically possible to model parts of an object oriented system functionally, which is quite an interesting observation and is definitely worth a more thorough approach.

In the next chapter we have a glance at *autoJAVA*, a plug-in for eclipse which was designed to integrate *Igor* into the eclipse workbench together with a simple way to provide input/output specifications to our system. The output of the system uses our simple protocol to generate 'quasi-object-oriented' notation.

5. AutoJava

Since this paper's focus is clearly on the theoretical part of how to design an object oriented program with a functional programming language it seems quite obvious not to get too much involved into the practical part of devising an application for this. However, *AutoJava* is a plug in for *eclipse* which basically provides a functionality to use *Igor* in an object oriented environment and as the focus of this paper is not on this prototype we are just going to have a short look at how a java file in this tool would look like:

Listing 14: Automated Solution of Last

```
/**
*@IgorMETA(
* methodName  = "last".
* retValue   = "Object".
* params     = "List".
*);
*@IgorEQ(
* equations = {
*     "([x])=x".
*     "([x,y])=y".
*     "([x,y,z])=z".
*     "([x,z,c,n])=n".
* }
*);
*@Method(last);
*/
public void last(){
    /***
    /* The following code has automatically been generated
         by AutoJava
    /* according to the user specification in the
         annotations above
    /* the result is printed below
    ***/

    //hypo(true, 2, eq 'Sub1['cons['X1:Object,'cons['X2:
         Object,
    //     'X3:List]]] = 'cons['X2:Object,'X3:List] [none] .
    //eq 'last['cons['X1:Object,'`['].List]] = 'X1:Object [
         none] .
    //eq 'last['cons['X1:Object,'cons['X2:Object,'X3:List
         ]]] = 'last['Sub1['cons[
    //     'X1:Object,'cons['X2:Object,'X3:List]]]] [none]
         .)

}
```

In this example we can see that java annotations [3] contain the specification which is considerably simplified from what we have seen so far. The most important parts are **IgorMETA** which contains method name, return value and input arguments of the method to infer. In the second bit *IgorEQ*, the input/output can be specified.

6. Conclusion

By now we have shown that it is possible to successfully model objects, methods, variables and messages in our simple protocol, moreover, we had igor synthesize all of them. So machine learning approaches have been used in order to have a system learn how to describe generic processes within programming languages. We provided a showcase of how functional programming can be combined with object orientation. The running plug in should prove this to be true and opens up many paths for future expansion.

A rather interesting point is the integration of the specification within the annotations which creates an entry point for large-scale applications such as IBMs *RSA*. As the developer can annotate his UML diagrams and have those annotations transferred into the auto-generated code you could think of a use case like Igor using the specification during the code generation filling in the method implementation.

All in all there has to be said that even though the results presented in this paper do not seem very novel or breathtaking. But they nevertheless show that by enabling functional programs to deal with object orientation we can play to the strengths of both paradigms. Even though it has been mentioned that our model does not claim to be complete or even fully correct - it feels like that we have created an inspiration for some next steps which might gradually improve the methodology and finally result in a larger scale prototype which actually produces Java code instead of functional programs.

[3] see `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`

A. Complete Listings

A.1 Maude Specifications and some Results

Note: *Igor* frequently produces more than one output hypothesis - for the sake of transparency only one of them is listed in our results sections.

Listing 15: Object

```
fmod OBJECT is

  *** Knowledge about how objects wrap variables and
      methods
  *** Uses 'IDENTIFYER-MATCH' in the way methods are
      called on an object
  *** The same goes for variable extraction

  sorts InVec Object Var Method VarList MethodList List
      ListEl NPException .
    subsorts Method < NPException .
    subsorts Var < NPException .
    subsorts List < VarList MethodList .
    subsorts ListEl < Var Method .
  sorts Identifier DType .
  sort MyBool .

  *** DT definitions
  *** list to store any value
  op [] : -> List [ctor] .
  *** object constructor, taking a list of variables & a
      list of methods together with an
  *** identifier for the object
  op ___ : Identifier VarList MethodList -> Object [ctor]
      .
  op met : Identifier DType -> Method [ctor].
  op var : Identifier DType -> Var .
  ops id1 id2 id3 : -> Identifier .
  op dt : -> DType .
  op exc : -> NPException .

  *** standard operations
  op cons : ListEl List -> List [ctor] .

  *** defined function names (to be induced, preds, bk)
      ***
  op mcall : Object Identifier -> Method [metadata "
      induce"] .

  var oid : Identifier .


  eq mcall( (oid [] []), id1 ) =  exc .
  eq mcall( (oid [] []), id2 ) =  exc .

  eq mcall( (oid [] cons(met(id1, dt), []) ), id1 ) = met
      (id1, dt) .
  eq mcall( (oid [] cons(met(id2, dt), []) ), id1 ) = exc
      .
  eq mcall( (oid [] cons(met(id1, dt), []) ), id2 ) = exc
      .
  eq mcall( (oid [] cons(met(id2, dt), []) ), id2 ) = met
      (id2, dt) .

  eq mcall( (oid [] cons( met(id1, dt), cons( met(id2, dt
      ), []) ) ), id1 )  = met(id1, dt) .
  eq mcall( (oid [] cons( met(id2, dt), cons( met(id1, dt
      ), []) ) ), id1 )  = met(id1, dt) .
  eq mcall( (oid [] cons( met(id2, dt), cons( met(id1, dt
      ), []) ) ), id2 )  = met(id2, dt) .

  eq mcall( (oid [] cons( met(id1, dt), cons( met(id2, dt
      ), cons( met(id3, dt), [])))) ), id1 )  =
    met(id1, dt) .

  eq mcall( (oid [] cons( met(id3, dt), cons( met(id1, dt
      ), cons( met(id2, dt), [])))) ), id1 )  =
    met(id1, dt) .

  eq mcall( (oid [] cons( met(id3, dt), cons( met(id2, dt
      ), cons( met(id1, dt), [])))) ), id1 )  =
    met(id1, dt) .

endfm
```

Listing 16: Identifier-Match

```
fmod IDENTIFIER-MATCH is

  *** Knowledge about how methods are called by providing
      an identifyer ***
  *** If a list of methods contains the called identifyer
      , the method is returned ***

  sorts InVec List Method Identifier DType ParamList
      NPException .
    subsort Method < NPException .

  op [] : -> List [ctor] .
  op cons : Method List -> List [ctor] .
  op mm : Identifier DType ParamList -> Method [ctor] .
  ops id1 id2 id3 : -> Identifier .
  op parlist : -> ParamList [ctor] .
  op exc : -> NPException .
  op dt : -> DType .


  op match : Identifier List -> Method [metadata "induce"
      ] .

  vars m1 m2 m3 : Method .

  eq match(id1, [] ) = exc .
  eq match(id2, [] ) = exc .

  eq match(id1, cons(mm(id1, dt, parlist) ,[]) )= mm(id1,
      dt, parlist) .
  eq match(id1, cons(mm(id2, dt, parlist), []) ) = exc .
  eq match(id2, cons(mm(id1, dt, parlist) ,[]) ) = exc .
  eq match(id2, cons(mm(id2, dt, parlist), []) ) = mm(id2
      , dt, parlist) .

  eq match(id1, cons(mm(id1, dt, parlist), cons(mm(id2,
      dt, parlist), [])) ) = mm(id1, dt, parlist) .
  eq match(id1, cons(mm(id2, dt, parlist), cons(mm(id1,
      dt, parlist), [])) ) = mm(id1, dt, parlist) .
  eq match(id2, cons(mm(id2, dt, parlist), cons(mm(id1,
      dt, parlist), [])) ) = mm(id2, dt, parlist) .

  eq match(id1, cons(mm(id1, dt, parlist), cons(mm(id2,
      dt, parlist), cons(mm(id3, dt, parlist), []))) ) =
    mm(id1, dt, parlist) .

  eq match(id1, cons(mm(id3, dt, parlist), cons(mm(id1,
      dt, parlist), cons(mm(id2, dt, parlist), []))) ) =
    mm(id1, dt, parlist) .

  eq match(id1, cons(mm(id3, dt, parlist), cons(mm(id2,
      dt, parlist), cons(mm(id1, dt, parlist), []))) ) =
    mm(id1, dt, parlist) .


endfm
```

Listing 17: Identifier-Match Result

```
eq: match(X1,()) = exc;
ceq: Sub1(X1,cons(mm(X2,dt,parlist),X3)) = X1 if == (X1,
    X2) = false;
ceq: Sub2(X1,cons(mm(X2,dt,parlist),X3)) = X3 if == (X1,
    X2) = false;
ceq: match(X1,cons(mm(X2,dt,parlist),X3)) = match(Sub1(X1
    ,cons(mm(X2,dt,parlist),X3)),Sub2(X1,cons(mm(X2,dt,
    parlist),X3))) if == (X1,X2) = false;
ceq: match(X1,cons(mm(X2,dt,parlist),X3)) = mm(X1,dt,
    parlist) if == (X1,X2) = true;
```

Listing 18: OO-Call

```
fmod OO−CALL is

  sorts InVec Object .
  sorts Message ParamList .
  sorts Nat Bool Param Res .
    subsorts Param < Nat Bool .
    subsorts Res < Nat Bool .
    subsorts Object < Nat Bool .


  *** DT definitions
  op * : −> Object [ctor] .
  op <> : −> ParamList [ctor] .
  op msg : ParamList Object −> Message [ctor] .
  op null : −> Object [ctor] .


  op 0 : −> Nat [ctor] .
  op s : Nat −> Nat [ctor] .
  op t : −> Bool [ctor] .
  op f : −> Bool [ctor] .

  *** Standard Operators ***
  *** op call : Message −> Message [metadata "pred␣
       nomatch"] . ***
  op cpar : Param ParamList −> ParamList [ctor] .


  *** defined function names (to be induced, preds, bk)
       ***
  op method : Message −> Message [metadata "induce"] .

  *** input encapsulation ***
  op in : Message −> InVec [ctor] .

  vars pl : ParamList .
  vars n : Nat .

  *** input output examples for "even" ***
  eq method( msg(cpar( 0,  <>), null) ) = msg(<> ,t ) .
  eq method( msg(cpar( s(0),  <>), null) ) = msg( <> , f
       ) .
  eq method( msg(cpar( s(s(0)), <>), null) ) = msg( <> ,
       t ) .
  eq method( msg(cpar( s(s(s(0))), <>), null) ) = msg( <>
       , f ) .
  eq method( msg(cpar( s(s(s(s(0)))), <>), null) ) = msg(
       <> , t ) .

endfm
```

Listing 19: OO-Call Result

```
eq: method(msg(cpar(X1,<>),null)) = msg(<>,f) if == (X1,s
     (0)) = true AND == (X1,s(0)) = true AND == (X1,s(0))
     = true;
ceq: method(msg(cpar(X1,<>),null)) = msg(<>,f) if == (X1,
     s(s(s(0)))) = true AND == (X1,s(s(s(0)))) = true AND
     == (X1,s(s(s(0)))) = true AND == (X1,s(s(s(0)))) =
     true AND == (X1,s(s(s(0)))) = true;
ceq: method(msg(cpar(X1,<>),null)) = msg(<>,t) if == (X1,
     s(s(s(0)))) = false;
```

Listing 20: Iterate-Collection

```
fmod ITERATE−COLLECTION is

  sorts Object Collection ResultCollection Method Result
       InVec .

  *** DT definitions (constructors)
  op [] : −> Collection [ctor] .
  op {} : −> ResultCollection [ctor] .
  op put : Object Collection −> Collection [ctor] .
  op put2 : Result ResultCollection −> ResultCollection [
       ctor] .
  op met : Object −> Result .
  *** defined function names (to be induced, preds, bk)
  op iterate : Collection −> ResultCollection [metadata "
       induce"] .
  *** input encapsulation
  op in : Collection −> InVec [ctor] .

  vars U V W X Y Z F : Object .


  eq iterate([]) = {} .
  eq iterate( put(Y,[]) ) = put2( met(Y), {}) .
  eq iterate( put(X,put(Y,[])) ) = put2( met(X), put2(
       met(Y) ,{})) .
  eq iterate( put(Y,put(X,put(Z,[]))) ) = put2( met(Y),
       put2( met(X), put2( met(Z),{}))) .

endfm
```

Listing 21: Iterate-Collection Result

```
eq: Sub1(put(X1,X2)) = met(X1);
eq: Sub2(put(X1,X2)) = iterate(Sub5(put(X1,X2)));
eq: Sub5(put(X1,X2)) = X2;
eq: iterate(()) = {};
eq: iterate(put(X1,X2)) = put2(Sub1(put(X1,X2)),Sub2(put(
     X1,X2)));
```

Listing 22: Foreach-Do

```
fmod FOREACH−DO is

  sorts InVec Object Var Method VarList MethodList List
       ListEl ParamList Collection .
    subsorts List < VarList MethodList .
    subsorts ListEl < Var Method .
    subsorts Object < Method .
  sorts Identifier DType .

  *** DT definitions (constructors) ***
  op [] : −> Collection [ctor] .
  op pp : −> ParamList .


  *** STANDARD OPERATORS ***
  op push : Object Collection −> Collection [ctor] .

  *** METHOD DECLARATION ***
  op call : Object Identifier DType ParamList −> Method [
       ctor].
  op id1 : −> Identifier [ctor] .
  op dt1 : −> DType [ctor] .

  *** defined function names (to be induced, preds, bk)
       ***
  op it_apply : Collection −> Collection [metadata "
       induce"] .

  *** input encapsulation ***
  op in : Collection −> InVec [ctor] .

  *** VARIABLES ***
  vars a b c : Object .


  *** ITERATION SPECIFICATION ***
  eq it_apply([]) = [] .

  eq it_apply( push(a, []) ) = push( call(a, id1, dt1, pp
       ), [] ) .

  eq it_apply( push(a, push(b, [])) ) =
    push( call(a, id1, dt1, pp), push( call(b, id1, dt1,
         pp), []) ) .

  eq it_apply( push(a, push(b, push(c, []))) ) =
    push( call(a, id1, dt1, pp), push( call(b, id1, dt1,
         pp), push( call(c, id1, dt1, pp), [])) ) .
```

endfm

References

Thomas Hieber. Transportation of the JEdit plug-in ProXSLbE to eclipse. Technical report, Otto Friedrich University of Bamberg, 2008. URL `http://www.cogsys.wiai.uni-bamberg.de/effalip/data/programs/autoXSL/ausarbeitung/projektbericht.pdf`.

Susumu Katayama. Systematic search for lambda expressions. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007. URL `http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/14num.pdf`.

Oleg Kiselyov and Ralf Laemmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005. URL `http://dblp.uni-trier.de/db/journals/corr/corr0509.html#abs-cs-0509027`. informal publication.

Emanuel Kitzelmann. Data-driven induction of recursive functions from I/O-examples. In Emanuel Kitzelmann and Ute Schmid, editors, *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 15–26, 2007.

Emanuel Kitzelmann. Analytical inductive functional programming. In *Pre-Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*. Michael Hanus, 2008.

Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006. ISSN 1533-7928.

Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995. URL `citeseer.ist.psu.edu/muggleton95inverse.html`.

Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990. URL `citeseer.ist.psu.edu/muggleton90efficient.html`.

Roland J. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1): 55–83, 1995.

J. Ross Quinlan and R. Mike Cameron-Jones. FOIL: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning,Proceedings*, volume 667, pages 3–20. Springer-Verlag, 1993. URL `citeseer.ist.psu.edu/quinlan93foil.html`.

Didier Remy and Jrme Vouillon. Objective ML: An effective object-oriented extension to ML, 1998.

Ute Schmid. *Inductive Synthesis of Functional Programs – Learning Domain-Specific Control Rules and Abstract Schemes*. Number 2654. Springer, 2003.

P. D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, 24(1):161–175, 1977.