Automated Method Induction: Functional Goes Object Oriented

Thomas Hieber and Martin Hofmann

University Bamberg - Cognitive Systems Group,
Feldkirchenstr. 21, 96050 Bamberg, Germany
`thomas-wolfgang.hieber@stud.uni-bamberg.de`,
`martin.hofmann@uni-bamberg.de`
`http://www.uni-bamberg.de/kogsys/`

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 translated into program code wherever possible. However, as few tools really go beyond this we demonstrate a simple functional representation for objects, methods and object-properties. In addition we show how our inductive programming system IgorII cannot only understand those basic notions like referencing methods within objects or using a simple protocol called *message-passing*, but how it can even learn them by a given specification - which is the major feature of this paper.

1 Introduction

IgorII is a system for synthesizing recursive functional programs, which learns potentially recursive functions solely from input/output (I/O) examples. Since IgorII is naturally based in functional programming, the main focus of this paper lies on finding a way to use IgorII for program inference in an object oriented background, which requires to express the behaviour 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 and as mainstream software for daily 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, *Inductive Programming* setting.

In addition, it is necessary to enable an object oriented programmer to provide input to the synthesis system as unobtrusive as possible. For this purpose, an interface for Eclipse 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)*. More practical concerns regarding the plug-in itself can be found in [1].

In this paper we will be introducing the concept of *Constructor Term Rewriting Systems (CTRS)* with respect to IgorII and how it can be used in order to construct a simple algebra to represent object orientation in a functional environment. After this we are going to show some examples to demonstrate how this can be done practically with the help of IgorII. The last chapter takes one step further explaining how this approach has been integrated in a prototype plug-in for the *Eclipse IDE*.

2 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' [2] paper on the induction of *LISP* programs. After the great success of *Inductive Logical Programming (ILP)* on classification learning in the nineties, research on IP shifted more to this area. Prominent ILP systems for IP are for example Foil [3], Golem [4] or Progol [5] - systems which make use of *Prolog* and predicate logic.

Later, the functional approach was taken up again by the analytical approaches IgorI [6], and IgorII [7] [8] and by the evolutionary/generate-and-test based approaches Adate [9] and MagicHaskeller [10].

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. Similar to classifier learning, *IP* systems can be characterised by a *preference* and a *restriction bias* [11].

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 [12] or OOHaskell ([13]). Additionally there are various approaches to include an object system in a functional language without changing the type system or the compiler (see e.g. [13] 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 naïve 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 Constructor Term Rewriting

For our purpose it shall be sufficient to define a functional program as a set of equations consisting of pairs of terms over a many-sorted signature Σ . We are going to adapt the common nomenclature as used in [14] when describing the concepts below using *terms* and *term rewriting*. A *signature* is defined as a set of function symbols Σ and a set of variables $\mathcal{X}$ which are used to form terms. In other words, terms over Σ and $\mathcal{X}$ are denoted $\mathcal{T}_\Sigma(\mathcal{X})$ whereas variable free terms (ground-terms) are just labelled $\mathcal{T}_\Sigma$. Since Σ is many-sorted, all our terms are typed.

One important differentiation to be alert of is that function symbols can either be datatype *constructors* or (user-)defined *functions*. In this fashion $\Sigma = \mathcal{C} \cup \mathcal{F}, \mathcal{C} \cap \mathcal{F} = \emptyset$, where $\mathcal{C}$ contains the constructors and $\mathcal{F}$ the defined function symbols. Our inductive programming system IgorII represents result programs as a set of recursive equations (rules) over a signature Σ. These rules consist of a left-hand side (lhs) and a right-hand side (rhs). While a rhs consists of regular Σ terms $\mathcal{T}_\Sigma(\mathcal{X})$, the lhs has the form $F(p_1, ..., p_n)$ and is called the *function head* with $F \in \mathcal{F}$ being the name of the function implemented by the current rewrite rule (plus some others). The $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$ consist of constructors and variables only.

$Var(t)$ are all variables of a Term t. The constructor terms p_i on the lhs of an equation may contain variables and are called *pattern*. All variables on the rhs of an equation are required to occur in the pattern on the lhs. We say that such variables are *bound* (or *unbound* otherwise). A *substitution* is a function $\sigma : \mathcal{X} \to \mathcal{T}_\Sigma(X)$. For our purpose, we write it in postfix and extend it to terms replacing all contained variables simultaneously. So $t\sigma$ is the result of applying the *substitution* σ to term t, i.e., applying σ to each $v \in Var(t)$. If $s = t\sigma$, then t is called a *generalization* of s and we say that t *subsumes* s and s matches t by σ, respectively. Given two terms t_1 and t_2 and a substitution σ such that $t_1\sigma = t_2\sigma$, we say that t_1 and t_2 *unify*. Given a finite set of terms $S = s, s', s'', ...$ then there exists a term t which subsumes all terms in S and which itself is subsumed by any other term also subsuming all terms in S. The term t is called the *least general generalisation* (lgg) [15] of the terms in S. To generalise a lggs to a set of equations, we tacitly treat the equal sign as a constructor symbol with the lhs and the rhs as arguments.

The operational semantics of a set of equations in the above mentioned form are best described in terms of a *term rewriting system* (TRS). An equation can be read as a *simplification* (or *rewrite*) *rule* replacing a term matching the lhs by the rhs. TRS which equations have the above described form are called *constructor term rewriting systems* (*CTRS*). From now on, we use the terms equation and rule as well as equation set and *CTRS* interchangeably throughout the paper, depending on the context. Let i be the vector $i_1, ..., i_n$. Evaluating an n-ary function F for an input i consists of repeatedly rewriting the term $F(i)$ w.r.t. the rewrite relation R implied by the *CTRS* until the term is in *normal form*, i.e., cannot be rewritten further. A sequence of (in)finitely many rewrite steps $t_0 \to_R t_1 \to_R ...$ is called *derivation*. If a derivation starts with term t and results in a normal form s this is written $t \overset{!}{\to}_R s$. We say that t normalises to s and call s the normal form of t. In order to define a function on a domain (a set of ground terms) by a CTRS, no two derivations starting with the same ground term may lead to different normal forms, i.e., normal forms must be unique. A sufficient condition for this is that no two lhss of a CTRS unify; this is a sufficient condition for a CTRS to be *confluent.* A CTRS is *terminating* if each possible derivation terminates. A sufficient condition for termination is that the arguments/inputs of recursive calls strictly decrease within each derivation and w.r.t. a well founded order.

Each rewrite rule may be augmented with a condition that must be met to apply the conditional rule. A term rewriting system or constructor system is called conditional constructor term rewriting system (CCTRS) respectively if it contains at least one conditional rule. A condition is an ordered conjunction of equality constraints $v_i = u_i$ with $v_i, u_i \in \mathcal{T}_(X)$. Each u_i must be grounded if the lhs of the rule is instantiated and if all equalities $v_j = u_j$ with $j < i$ evaluate to true, then u_i evaluates to some ground normal form. For the v_i must hold (i) either the same as for the u_i or (ii) v_i may contain unbound variables but then it must be a constructor term. In the first case also v_i evaluates to some ground normal form and the equality evaluates to true if both normal forms are equal. In the second case the equality evaluates to true if v_i and the ground normal form of u_i unify. Then the free variables in v_i are bound and may be used in the following conjuncts and the rhs of the rule. We write conditional rules in the form: $l \rightarrow r \Leftarrow v_1 = u_1, ..., v_n = u_n$. The ! indicates negation, thus $!(v_i = u_i)$ holds if both normal forms do not unify. Rules without a condition are called unconditional. If we apply a defined function to ground constructor terms $\mathcal{F}(i_1, ..., i_n)$, we call the i_i inputs of $\mathcal{F}$. If such an application normalises to a ground constructor term o we call o output. A CCTRS is terminating if all rewriting processes end up in a normal form. In order to implement functions the outputs are required to be unique for each particular input vector. This is the case if the TRS is confluent.

4 Igor

IgorII is a prototype for constructing recursive functional programs from few non-recursive, possibly non-ground example equations describing a subset of the input/output (I/O) behaviour of a function to be implemented. For all of the example specifications in IgorII the signatures of the following functions shall be established:

$$[\,] \; : \rightarrow List \tag{1}$$

$$cons \; : \; Element \times List \rightarrow List \tag{2}$$

$$s \; : \; Nat \rightarrow Nat \tag{3}$$

$$0 \; : \rightarrow Nat \tag{4}$$

$$t \; : \rightarrow Boolean \tag{5}$$

$$f \; : \rightarrow Boolean \tag{6}$$

Here is a simple example of how the list-function *even* would be presented to the system as I/O examples. Please note that constructor symbols and function names are in lower case, variables in upper case.

```
even(0) = t
even(s(0)) = f
even(s(s(0))) = t
even(s(s(s(0)))) = f
even(s(s(s(s(0))))) = t
```

The induction of a correct program in IgorII is organised as a best-first search. During a search, a hypothesis is a set of equations entailing the example equations but potentially with unbound variables in the right-hand side. Starting from an initial hypothesis, successively the best hypothesis, w.r.t. some preference bias, is selected and an unfinished rule is chosen and replaced by its successor rules. This is continued until the current best hypothesis does not contain any unbound variables.

Initial Rule. The initial hypothesis contains one rule per target function. This rule is a *least general generalisation* (lgg) of the example equations. The lgg for the previous *even*-examples is:

```
even (N) = B
```

Without getting into theoretical details, it should be sufficient to know for now that constructor symbols or sub-terms occurring at the same position in all equations are kept, everything else is substituted by variables. We say, that the rule covers all previous examples, because the pattern on the lhs subsumes each lhs of the examples. Of course, this rule is not a functional program, because it contains an unbound variable on the rhs. To remedy this, the initial hypothesis is stepwise re-defined. For this purpose, IgorII employs three transformation operators:

1. The I/O examples belonging to the open initial rule are partitioned into subsets and for each subset, a new initial rule (with a more specific pattern, or left-hand side, than the original rule) is computed.
2. If the open rhs has a constructor as root, i.e., does not consist of a single unbound variable, then all sub-terms containing unbound variables are treated as sub-problems. A new auxiliary function is introduced for each such sub-term.
3. The open right-hand side is replaced by a (recursive) call to a defined function. The arguments of the call may be computed by new auxiliary functions. Hence, computing the arguments is considered as a new sub-problem.

Splitting an open rule. The first operator partitions the I/O examples belonging to a rule into subsets such that the patterns of the resulting initial rules are disjoint and more specific than the pattern of the original rule. Finding such a partition is done as follows:

A position in the pattern p with a variable resulting from generalising the corresponding sub-terms in the subsumed example inputs is identified. This implies that at least two of the subsumed inputs have different constructor symbols at this position. Now all subsumed inputs are partitioned such that all of them with the same constructor at this position belong to the same subset. Together with the corresponding example outputs this yields a partition of the example equations whose inputs are subsumed by p. Since more than one position may be selected, different partitions leading to different sets of new initial rules may result.

Introducing (Recursive) Function Calls and Auxiliary Functions. In cases (2) and (3) auxiliary functions are invented. This includes the generation of I/O-examples from which they are induced. For case (2) this is done as follows: Function calls are introduced by matching the currently considered outputs, i.e., those outputs whose inputs match the pattern of the currently considered rule, with the outputs of any defined function. If all current outputs match, the rhs of the current unfinished rule can be set to a call of the matched defined function. The argument of the call must map the currently considered inputs to the inputs of the matched function. For case (3), the example inputs of the new defined function also equal the currently considered inputs. The outputs are the corresponding sub-terms of the currently considered outputs.

Terms matching the lhs of a rule, where a variable can subsume any sub-term of the accordant type, can be replaced by the rhs of this rule. This procedure is repeated until the term does not match any more lhs.

Example. Using the data type specification and the I/O examples given below, we will sketch the IgorII algorithm by developing a solution for the list-function *even*. Starting from the initial rule

```
even(N) = B
```

The IgorII algorithm successively develops a solution that is correct and complete w.r.t. the I/O examples using the previously described operators.

In the first step, the example set is partitioned w.r.t. root constructor symbol in the first argument.

```
even(0) = t
even(s(N)) = B
```

The second rule now covers all examples but the first one. In a second step, again a partition is introduced. Now the constructor symbol of the first subterm below the root position discriminates the I/O examples.

```
even(0) = t
even(s(0)) = f
even(s(s(N))) = B
```

The first and the second rule are the base cases of the target function, both covering only one I/O example. The third rule, becoming the recursive call, covers the rest. Following the second partition of the rule set, an auxiliary function is introduced since the rhs contains a constructor symbol:

```
even(0) = t
even(s(0)) = f
even(s(s(N))) = sub1(s(s(N)))
sub1(s(s(N)) = even(N)
```

This result computed by IgorII is a correct and complete w.r.t. the I/O examples, recursive solution to the problem.

Up to this point it should be clear how in the context of a CTRS, IgorII develops a correct functional solution from a set of non–recursive I/O examples. What is left to do now is how we can manage to encapsulate the functional flavour in an object oriented protocol since we are trying to bring those two paradigms together.

5 Igor and Object Orientation

In order to model object oriented processes in a functional way we are going to use two *Constructor Term Rewriting Systems (CTRSs)*. This is done for quite a simple purpose: *encapsulation*. One *CTRS* will be employed to model object orientation as a simple protocol, the other one will be used to encapsulate the problem domain as described in section 3. This is as much as we need to understand for now as we are going to come back to that later on.

The two *CTRSs* are going to be defined like this:

$$C_{OO} : \Sigma = \mathcal{F} \cup \mathcal{C} \cup \{\mathcal{D}\}, \mathcal{X}, \mathcal{E} \tag{7}$$

$$C_P : \Sigma' = \mathcal{F}' \cup \mathcal{C}', \mathcal{X}', \mathcal{E}' \tag{8}$$

The first equation describes the object oriented protocol which will be introduced in this section. One main difference to the second one has to be remarked, since $\mathcal{D}$ represents a *constant* which will be used as place holder for terms over C_{OO}

The second equation defines the *CTRS* describing terms in the problem domain. They shall be related to each other in a way that C_P is a proper Sub CTRS of C_{OO}.

The concepts *Super Constructor Term Rewriting System* (C_{OO}) and *Sub Constructor Term Rewriting System* (C_P) shall be defined like this:

Definition 1. C_P is a Sub CTRS of C_{OO} ($C_P \subset C_{OO}$), iff

$$\begin{aligned} \Sigma' \subset \Sigma \; s.t. \; \mathcal{F}' \subset \mathcal{F} \; and \; \mathcal{C}' &\subset \mathcal{C} \\ \mathcal{X}' &\subset \mathcal{X} \\ \mathcal{E}' &\subset \mathcal{E} \end{aligned}$$

Definition 2. C_{OO} is a Super CTRS of C_P, iff

$$C_P \subset C_{OO}$$

5.1 The Super CTRS

Since the aim of this paper is to define an algebraic definition of a simple object oriented protocol, the major part of this section is concerned with the *Super CTRS* and how it models object orientation. Before we proceed it is important to point out that it is not in the focus of this work to create a full-scale model of object orientation. Rather have we singled out a couple of interesting mechanics and put them together to a tiny fragment of object orientation, the one which is concerned with identifying methods and properties in an object and interacting with them.

For this we establish the already known concepts of methods and properties (a.k.a. member-variables) along with a protocol we call *message-passing*, which is used by objects in order to interchange data. The relevant parts contained in the protocol will now be described by relating them to Σ of C_{OO}. For this we need the constant $\mathcal{D}$, constructors $\mathcal{C}$ and the functions $\mathcal{F}$ themselves.

$$data :\to Data \tag{9}$$

Data will be treated as constant throughout the object oriented protocol, since it is used to have it as wild card for terms of the problem domain it should be self evident that it strictly consists of terms in C_P, in other words $T_{\Sigma'}(\mathcal{X}')$. This is also the actual trick which makes the two domains independent of each other, which should become clear in the course of this section.

The definition of an object in our algebra would look like this:

$$object : \ Identifier \ \times \ PropList \ \times \ MethodList \ \to Object \tag{10}$$

The constructor's arguments are the object's *Identifier*, a *List* of properties (object resident member variables) and a *List* of (object resident) methods. Later on it is going to be of importance whether an object contains a method or not - this is specified with the help of this constructor.

As the object constructor needs a list of properties to be provided it is time to find out how they can be constructed.

$$property : \ Identifier \ \times \ Data \ \to Property \tag{11}$$

Just like before, a property must be labelled with an *Identifier*, the value on the other hand seems rather obscure. *Data* is used to abstract the information contained within the property. This is a crucial section in this protocol since it is the 'wrapper' for our *Sub CTRS*. So, whenever *Data* is used it should be clear that it is the 'packaging' for terms of our problem domain and the only way to combine it with the object oriented protocol. Because it is intended to draw a clear distinction between the two *CTRSs* it is important to understand that using *Data* on the level of object oriented communication is more than enough and all to be aware of at this stage.

There are two more constructors left to define, the first is the one for a *Method*:

$$method : \ Identifier \times Message \to Method \tag{12}$$

As it is already evident that a method will have to be identified later on, an *Identifier* needs to be declared for it as done for *Object* and *Property*. The second argument is called *Message* and this is the important part in our protocol when it comes to object interaction. As soon as an object needs to call a method or get the value of a property on any other object (or even itself), it will send a *Message* which contains some kind of data.

The nature of the data becomes clear looking at the constructor of the *Message*:

$$message : \ Data \to Message \tag{13}$$

As seen in the *Property* constructor the actual nature of the data transported between objects is abstracted. And this should be quite self evident now since it has been the intention to keep the object oriented protocol strictly apart from the actual data processed with it's help. It is not necessary to know what is inside *Data*, since the only concern for now is how to transport it from one object to another.

This brings us to the concept of *message-passing*. The idea behind it is quite literal the exchange of messages, whenever we are trying to access an object's property or call one of it's methods. Before it is possible to call a function or a property's value they have to be looked up in the target objects' property list/method list.

This can be done by the two following functions:

$$Match\_method : \ Identifier \times MethodList \to Method \tag{14}$$

$$Match\_prop : \ Identifier \times PropList \to Property \tag{15}$$

For convenience, things are sped up now since it is not hard understanding how to access to the two lists containing the target object's methods/properties. And for now it is not a problem as it is our main focus to clarify how to access a method or a property given a random object. Those functions should demonstrate that a positive match of identifiers within an object's method/property list will return the method/property we are trying to address and raise an exception otherwise.

All that is left to do is to request either the value of it, or call it with a list of method arguments.

It has just been mentioned that the part where the method/property list from an object are extracted - so this is the formal approach in our protocol:

$$Call : \ Object \times Identifier \times Message \to Message \tag{16}$$

Up to now the development of signatures in C_{OO} has been delivered and it is intriguing to find out how to proceed further, since the protocol is far away from being complete. The major part of the work left to do will be carried out by IgorII, who will take those signatures and induce the according functions with the help of some simple I/O examples.

5.2 The Sub CTRS

As already pointed out, the *Sub CTRS* is entirely separated from the object-oriented message protocol. This means that it can encapsulate virtually *anything* without interfering with the *Super CTRS*. And this is where the beauty of our approach lies in since we have now successfully separated the way to represent a problem domain structurally from the way it is represented semantically. Since IgorII can only understand functional problem specifications it seems quite rational to have them represented using only terms over a functional algebra – in this case it is C_P.

6 Examples

In section 5 the introduction of two *CTRSs* has been used in order to define a simple object-oriented protocol encapsulating a functional problem domain. However, only signatures have been defined by now. And in order to receive the mechanics of the object-oriented methods like *Match_method*, *Match_prop* and *Call* the inductive programming system IgorII is going to be used to infer those methods just from a few I/O examples. When constructing I/O examples for the system, we use the signatures defined in section 5, which we will include again below:

$$Match\_method : Identifier \times MethodList \rightarrow Method \tag{17}$$

$$method : Identifier \times Message \rightarrow Method \tag{18}$$

The first method to infer is *Match_method*, however, in order to express an unsuccessful match, an *exception* shall be defined beforehand in the following manner:

$$exc : \rightarrow Method \tag{19}$$

Mind that the constant *exception* has not been defined earlier, it is just a placeholder for any kind of imaginable procedure to capture a 'no-match'. Apart from that, the *matching* process is passed a variable as identifier as well as a list of methods which of course would have been taken from an existing object. In case the requested identifier is matched by a resident method within the method list, the according method is returned, otherwise an exception is thrown.

The examples for the method itself read as follows:

```
match_method(Id1 []) = exc
match_method(Id2 []) = exc

match_method(Id1 cons(method(Id1 msg) [])) = method(Id1, msg)
match_method(Id1 cons(method(Id2 msg) [])) = exc
match_method(Id2 cons(method(Id1 msg) [])) = exc
match_method(Id2 cons(method(Id2 msg) [])) = method(Id2, msg)

[...]
```

IgorII takes those examples and constructs the following set of equations:

```
 1) match_method(Id1 []) = exc

 2) sub1(Id1 cons(method(Id2 msg) Restlist)) =
    Id1 <== !(Id1 = Id2)

 3) sub2(Id1 cons(method(Id2 msg) Restlist)) =
    Restlist <== !(Id1 = Id2)

 4) match_method(Id1 cons(method(Id2 msg) Restlist)) =
    match_method(sub1(Id1 cons(method(Id2 msg)
    sub2(Id1 cons(method(Id2 msg) Restlist) ))))
    <== !(Id1 = Id2)

 5) match_method(Id1 cons(method(Id2 msg) Restlist) ) =
    method(Id1 msg) <== (Id1 = Id2)
```

The same result is returned for *Match_method*, which will be left out here since it is almost identical to *Match_method*. Moving one step further we are going to find out whether the *Call* method can be induced as easily - the answer is yes!

Remember the signature of *Call*:

$$Call : Object \times Identifier \times Message \rightarrow Message \tag{20}$$

Now it has been demonstrated that IgorII can not only understand, but also induce the simple object—oriented protocol which is reason enough to try and insert some terms from the problem domain into our specification. For this the example from section 4 is used, so the functional I/O examples for *even* will be encapsulated within an object oriented specification. As the signature requests an actual *Object* to be part of the call, it is going to be inserted as the variable *O* of the type *Object*. This is enough for our purpose here, since it is never used and remains constant within all our examples.

```
Call(O even message(0)) = message(t)
Call(O even message(s(0))) =  message(f)
Call(O even message(s(s(0)))) = message(t)
Call(O even message(s(s(s(0))))) = message(f)
Call(O even message(s(s(s(s(0)))))) = message(t)
```

The result is as straightforward as before, but notice how the *message-passing* is used consistently:

```
  1) call(O even message(0)) = message(t)

  2) call(O even message(s(0))) = message(f)

  3) call(O even message(s(s(X1)))) =
     call(sub19 message(s(s(X1))))

  4) sub19(message(s(s(X1))))  = message(X1)
```

Here it is clearly evident that the object-oriented terms do not obstruct IgorII in any way, an explanation as to why will be given shortly. Another more complex example has the system induce an operator which is very famous in object orientation - the *iteration*. For the sake of readability those examples are not wrapped in messages and within calls. That this is no big problem will be clear at the end of this section, so bear with us for now and just look at the example:

```
iterate([]) = []
iterate(cons(A [])) = cons(call(A))
iterate(cons(A cons(B []))) = cons(call(A) cons(call(B) []))
iterate(cons(A cons(B cons(C [])))) =
cons(call(A) cons(call(B) cons(call(C) [])))
```

The very abstract idea is that a rather abstract *Call* (which could be any defined method) is applied to every object within a list. As expected, IgorII comes up with a recursive solution to this problem:

```
1) iterate([]) = []

2) iterate(cons(Object Restlist)) =
   cons(sub1(cons(Object Restlist)) sub2(cons(Object Restlist)))

3) sub1(cons(Object Restlist)) = call(Object)

4) sub2(cons(Object Restlist)) =
   iterate(sub5(cons(Object Restlist)))

5) sub5(cons(Object Restlist)) = Restlist
```

As mentioned before have we reduced the complexity of our *Call* signature in order to increase readability of the examples. One question when dealing with all the object-orientation might have been urging all the time: *Isn't this a lot of overhead?*

Therefore we are going to combine the signatures of *Object*, *Call*, *Message* and *Method*, generating a *Call* of the signature:

$$Call : Object \times Identifier \times Message \to Message \tag{21}$$

Into the call (lines 1–8), an *Object* (line 3) is inserted, containing an identifier *OId*, an (empty) property list and two methods *even* and *odd* (all in line 3). When we would now call the function *even* (line 5) with a *Message* containing a singleton list as argument (line 7) it would look like this:

```
 1: (
 2:   # <-- Object: Identifier x PropList x MethodList
 3:   (OId [] cons((even bool ([] [])) (odd bool ([] []))))
 4:   # <-- Identifier
 5:   even
 6:   # <-- Message
 7:   message(0
 8: )
 9:  # <-- Message
 10: = message(t)
```

This looks very bloated and one might wonder if all this is prone to slow down IgorII during the synthesizing? For this let us have a look how it anti-unifies the terms gradually:

```
(
  (OId [] cons((even bool ([] []) ) (odd bool ([] []))))
  even
  message(succ(0))
)
= message(f)

(
  (OId [] cons((even bool ([] [])) (odd bool ([] []))))
  even
  message(succ(succ(0)))
)
= message(t)
```

after anti-unification:

```
 (
   (OId [] cons((even bool ([] [])) (odd bool ([] []))))
   even
   message(succ(t))
 )
```

Looking at those examples you find the answer to the question if the protocol's overhead is impeding the system and at the same time you get the key aspect of all the examples and the increasing complexity: **it doesn't matter**! From this it will also be plain to understand why the iteration example has been kept quite simple with no *message-passing* involved at all. It would just be additionaly code around the problem specification itself. And that is the one which matters to IgorII, since it does not even use the terms from C_{OO} in any way.

7 AutoJAVA

All the theory so far has clarified the fact that a functional inductive programming system as IgorII can be confronted with problems outside the functional context. You may have noticed that we are still delivering the problems in a functional way, since this is the only format the system can understand. But it should be obvious that after successfully modeling a very small protocol of object oriented flavour we could enlarge this tiny model into a larger one, much more like a real object oriented protocol.

In order to use our findings in a practical way, a prototype plug-in for eclipse was designed which was aimed to enable a programmer to characterise the behaviour of a function in an abstract way (I/O examples) and having IgorII create a program from this specification, which would be wrapped into our C_{OO}[1].

8 Conclusion

Not only have we successfully modelled objects, methods, properties and messages - we also 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 prototype plug-in *AutoJAVA* should prove this to be true and opens up many paths for future expansion. As the problem specification can be embedded within the annotations of a program's method an entry point for large-scale applications such as IBMs *RSA* has been created. A developer can annotate his UML diagrams and have those annotations transferred into the auto-generated code, so you could imagine Igor using the specification during the code generation filling in a method's 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. It feels like that we have created a foundation for some more thorough steps which might gradually improve the methodology and finally result in a larger scale prototype which actually produces Java code instead of functional programs.

[1] See more on `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`

References

1. Hieber, T.: Transportation of the JEdit plug-in ProXSLbE to eclipse. Technical report, Otto Friedrich University of Bamberg (2008)
2. Summers, P.D.: A methodology for LISP program construction from examples. Journal of the ACM 24(1), 161–175 (1977)
3. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Brazdil, P.B. (ed.) ECML 1993. LNCS, vol. 667, pp. 3–20. Springer, Heidelberg (1993)
4. Muggleton, S., Feng, C.: Efficient induction of logic programs. In: Proceedings of the 1st Conference on Algorithmic Learning Theory, Ohmsma, Tokyo, Japan, pp. 368–381 (1990)
5. Muggleton, S.: Inverse entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming 13(3-4), 245–286 (1995)
6. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7, 429–454 (2006)
7. Kitzelmann, E.: Data-driven induction of recursive functions from I/O-examples. In: Kitzelmann, E., Schmid, U. (eds.) Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP 2007), pp. 15–26 (2007)
8. Hofmann, M., Kitzelmann, E.: I/o guided detection of list catamorphisms – towards problem specific use of program templates in ip. In: Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM 2010) (to appear, 2010)
9. Olsson, R.J.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–83 (1995)
10. Katayama, S.: Systematic search for lambda expressions. In: van Eekelen, M.C.J.D. (ed.) Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, vol. 6, pp. 111–126. Intellect (2007)
11. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education, New York (1997)
12. Rémy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML (1998); A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages (1997)
13. Kiselyov, O., Laemmel, R.: Haskell's overlooked object system. CoRR (2005); informal publication
14. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
15. Plotkin, G.: A note on inductive generalisation. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 5, pp. 153–163. Edinburgh University Press, Edinburgh (1969)