Otto-Friedrich-Universität Bamberg

Cognitive Systems Group

Ausarbeitung

Bachelor Arbeit

Zum Thema:

Example-Driven Programming – A Tool for Automated Method Induction in Eclipse Based Environments

Vorgelegt von:

Hieber, Thomas

Betreuer: Prof. Ute Schmid

Bamberg, WiSe 2008/2009

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. Over the years the tools for software engineering have gradually been improved on. Today it is a standard to have UML diagrams be translated into program code wherever possible. Since this stops most commonly at the point of generating empty methods, the question arises if the technical development is ready to do the next step. Automated software engineering as a discipline closely connected to machine learning and AI in general. This Thesis focuses on the way a functional program induction system can handle object oriented input and how it could be used to help a programmer automatically generating method bodies given specific input/output examples. Together with the analysis of a case study from a student project, a prototype plugin for Eclipse-based environments is created and tested on the demands that arise from the analysis taken.

ERKLÄRUNG

Ich erkläre hiermit nach gemäß §17 Absatz 2 APO, dass ich die vorstehende Bachelor Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Bamberg, 30.03.2009

Thomas Hieber

Contents

1 **Software Engineering and Example Driven Programming** **1**

2 **Object Oriented or Functional?** **3**

2.1 Object Oriented Programming 3

2.2 Functional Programming 4

2.3 Differences 4

3 **Igor and Maude** **6**

3.1 Inductive Functional Programming 6

3.2 Genetic/Evolutionary Programming 6

3.3 Igor 7

4 **Case Study: Software Engineering** **8**

4.1 Assignment Outline 8

5 **Igor and Object Orientation** **10**

5.1 Representing the Object 10

5.2 Representing the Method 11

5.3 Representing the Method Call 11

5.4 Concerning Variables 14

5.5 Messages 15

5.6 Back Into Perspective 16

6 **Plug-in for Eclipse: autoJava** **19**

6.1 Requirements & Installation 19

6.1.1 Some Notes 19

6.2 autoJava in Theory 19

6.2.1 Antlrworks 22

6.2.2 Easyigor 23

6.2.3 Autojava 25

6.3 autoJava in Practise 25

6.4 autoJava in Conclusion 29

7 Watch what we did 30

8 Appendix 33

8.1 Maude Codesamples 33

8.2 Grammars 46

List of Figures

1 autoJava Architecture 20

2 autoJava List View with Context Menu 25

3 Syntax Error 25

4 Syntax OK 25

List of Tables

1 Group 1 Statistics 8

2 Group 2 Statistics 8

Listings

1 Input/Output Examples for Igor . . . 7

2 Recursive Program for length . . . 7

3 Object Constructor . . . 11

4 Method Constructor . . . 11

5 Identifier Match . . . 12

6 Constructors and Variables . . . 12

7 Input/Output Examples . . . 13

8 Variable Constructor . . . 14

9 Message Constructor . . . 15

10 Identifier Match Sorts . . . 15

11 Identifier Match Constructors . . . 15

12 Identifier Match Input/Output Examples . . . 16

13 Iterate Collection Input/Output Examples . . . 17

14 Correct Annotated Specification . . . 22

15 Automated solution of Last . . . 26

16 Iteration over a collection . . . 27

17 To-Lower Specification . . . 28

18 Object . . . 33

19 Object Result . . . 34

20 Identifier-Match . . . 38

21 Identifier-Match Result . . . 39

22 OO-Call . . . 40

23 OO-Call Result . . . 40

24 Iterate-Collection . . . 41

25 Iterate-Collection Result . . . 41

26 Foreach-Do . . . 42

27 Foreach-Do Result . . . 42

28 To-Lower Result 43
29 Maude Template 44
30 Code sample from the implementation of group 2 45
31 Parenthesised Expression 46
32 Igor Annotation 47

1 Software Engineering and Example Driven Programming

The concept of *End-User Programming* formulated by Henry Lieberman ([9]) or Allen Cypher ([3]) was based on the assumption that the user be 'at the end of the process of computer programming' ([3]). Ever since the arrival of personal computing this has softened up to the point that nowadays in many cases the end user is a programmer. This aspect carries a whole new set of opportunities since the way programmers understand and use computer applications differs from the way a traditional user does. As Lieberman and Cypher put great effort into making *End-User Programming* accessible to anyone we might ask ourselves what would be possible if we wanted to enable a programmer to do the same thing. For sure we would not have to be as distant from real programming as we have to be with non-programmers. But what would we gain - and what for?

Lieberman describes the motivation for *End-User Programming* quite to the point: 'It is a truism that computers are good at performing repetitive activities. So why is it that *we* are the ones performing all of the repetition, instead of the computer?' This is quite sound and when we think about programming there seem to be many cases in which a programmer would like the computer do the work for him in order to focus on a set of problems he must try and solve. Just like it was at the the times when platforms like *Eclipse* weren't around or still very limited and (object oriented) programmers had the distinct pleasure of writing a great number of getters and setters before they could begin with the actual program.? Of course, today we are way past this point but now we might wonder if we could take the next step and use methods of machine learning in order to let the computer help us with some of the programming. So if there are still tasks which are repetitive but not as easily automated as the getter/setter solution it would be nice if we could have the programmer provide a rough idea of what he intends to let the computer handle the rest. Imagine if you would have to write a large switch-statement with a few alterations in every case, everyone who has done this will agree that it would be a relief not to be delayed by endless copy and paste actions with some minor alterations along the way (which might even contain errors at the end of the day). Or how about a problem which the programmer knows to be solved recursively but somehow he cannot get it done?

This thesis was at first planned to start off with an actual software project in IBM's *Rational Software Architect (RSA)* [1]. The idea was to give the programmer the possibility to annotate certain methods even before the actual coding with input/output examples in the tradition of *Example-Driven Programming.* Since this could happen as early as the modelling process takes place the programmer would then not only find *RSA* generate empty method bodies but some of them implemented already according to his example-specification.

As the proprietary standard of *RSA* doesn't allow access to the *UML* structure and the code building process, the task had to be slightly adjusted. Since the underlying framework is *Eclipse* it seemed plausible to try and find an entry point which would be general enough to be supplied by *RSA* without too much hassle. As it is possible in *RSA* to make notes on any *UML* diagram and have the system put them into the documented comments for each method, we could assume that this was our starting point and take it from there. So the task is to generate a system which would take a specification consisting

[1] see `http://www-01.ibm.com/software/awdtools/architect/swarchitect/`

of input/output examples and generate a working program. The main focus of this thesis does not lie on the inductive-programming part as such, we will rather try and bring object oriented and functional programming together, since the system used for program inference - *Igor* - produces only functional programs. So there will be a general comparison of the two paradigms, an introduction of the way *Igor* works along with an attempt to test it for its capability of working object oriented. Further, there will be a description of the implemented plug-in, *autoJava*, as well as an attempt to bring it together with a real software project and an evaluation thereafter.

2 Object Oriented or Functional?

As already pointed out, we will be concerned with the concepts of *functional programming* and *object oriented programming*, especially with their features regarding the translation of object oriented concepts into the functional world.

Let us therefore now do some basics and describe the two paradigms and the differences between them. In [2] the family of *high level programming languages* is split up into the *procedural* and *non-procedural* branch. To distinguish them from each other, the way they work is described as either a description of *how* a result is accomplished (procedural) or '*what* is to be accomplished'. So while the former is more like a recipe to solve a problem, the second one can be regarded as a more abstract specification of the problem itself.

Now lets have a look at two concepts in procedural and non-procedural programming.

2.1 Object Oriented Programming

In its core, object oriented programming is procedural, just like the programming language *C*. When you look at the *main* method in modern object oriented languages like *Java*, *C++* or *C#* you will always find a list of procedures which are carried out one after another (forgetting about concurrency for one second). The innovation object orientation has brought to procedural programming is the encapsulation of program code within *objects*, taking *modules* to the next level. Running an object oriented program is in essence a number of objects interacting with each other. One key concept in this interactive process is inheritance. Objects can be related to each other with parent-child relationships thus saving time and lines of code since inheritance ensures that all the features from a parent object are available in the child. The major notion for the specification of objects is *class*. At runtime, classes are instantiated and then become objects. The most common language constructs used are iterations and conditionals like the well known *while-loop* or *if-then-else* statements. Encapsulation, information hiding ensure a new level of abstraction. For matters of software engineering this becomes all the more valuable since business processes and entities can now be directly converted into code, as well as their states and behaviours. Concepts like *design classes* or *business objects* ([1]) are state-of-the-art when it comes to software engineering which is closely connected to object orientation in every aspect from the modelling (*UML*) right down to the implementation (object oriented programming). So for the *famous three (C++, Java, C#)* out of this genus it is well justified to use the term 'productive language' as defined by Cezzar ([2]).

2.2 Functional Programming

The name of this paradigm gives us the centre piece and the all important component - the *function*. While the notion of a 'function' occurs in many other programming languages, which are not necessarily functional, it is important to understand that they are very different from each other. While a function in a procedural program can have more than just one output, it cannot in a functional one. Since the functions here are strictly bound to the mathematical definition of a function, you will understand that there can be no such thing as:

$$f(x) = a, f(x) = b; a \neq b$$

This means that there cannot be a global state within a functional program since the value of an expression must always remain the same, even though its evaluation may change the form of expression. This feature is called *referential transparency* ([4]). Functions can be composed in order to build more complex functions, treating inner functions as black boxes, the technical term for this is *functional decomposition*. One of the most basic procedures in functional programming is *reduction*. This is very intuitive and directly follows mathematical concepts dealing with equations. Everything can be reduced to its *normal form* like this:

$(1 + 2)^2 \Rightarrow (1 + 2) * (1 + 2) \Rightarrow 3 * 3 \Rightarrow 9$

One last and very powerful tool in the functional paradigm are *high order functions*. This means that any function can have another function as argument or as result output which provides a crucial mechanism for the programmer to use. The *map* function is a prominent example for *high order functions*. More features like *pattern matching*, *type inference* or *lazy evaluations* are illustrated in [8]. Of course, everything in functional programming revolves around *recursion*. The different variations like *tail recursion* and the like are most important in order to create a functional program which is effective and 'to the point'.

2.3 Differences

The list of differences between such vastly distinguished paradigms is of course endless. For our purpose it is enough to keep two things in mind. While one allows for global program states (object oriented programming), the other (functional programming) doesn't. Even more important is the second aspect - the use of recursion. While recursion is generally possible in object orientation it is not always necessary. Most of the time programmers use simple iterations and get along just fine. This is not the case in the functional world. Many problems cannot be solved without the use of recursion, it is deep-seated within the concept in many ways. The definition of lists, for example, or the structured definition of the natural numbers using the *successor* operator can be taken as evidence. This will be a very important point when we get to our case study later on (chapter 4) and we will see that in the most unfortunate cases there is no need for recursion in the object oriented world at all.

Talking about software engineering, it has been mentioned that object orientation fits very elegantly into state-of-the-art technologies because it supports encapsulation, inheritance and information hiding. Purists may now claim that functional programming supports the very same concepts like enrichment or abstraction and any specification can very well be written down functionally. Even though this is the truth it is still not as intuitive to write productive software in a functional background - just remember the fact that it is not possible to have global states and such.

Before we consider our case study let us first have a look at the way recursive functional programs can be induced from input/output pairs. This is - as you will remember - our task in *autoJava* (chapter 6) and has to be properly introduced before we move on.

3 Igor and Maude

As already pointed out, we will later on be concerned with the automated induction of programs. For this it is important to understand the way program induction works and which techniques it is based on.

3.1 Inductive Functional Programming

The extraction of programs from input/output examples has been around since the seventies, this sub-branch *Inductive Logic Programming (ILP)* has been greatly influenced by Summers' ([12]) paper on the induction of *LISP* programs. There have been other approaches like *Inductive Logical Programming (ILP)* with systems like *FOIL* [2] or *GOLEM* [3], systems which make use of *Prolog* and predicate logic. All in all you can subsume the concern of *Inductive Programming* as the search for algorithms which use as little information as possible to generate correct computer programs from a given minimal specification.

3.2 Genetic/Evolutionary Programming

In the last years there has been a new fish in the pond which is referred to as *genetic* or *evolutionary* approach. This method resembles the way of nature on the development of inferred program fragments. Other than the *IP* approaches the hypotheses are not really generated by strategy, but randomly if you will. Like biological evolution, programs are mutated and evaluated until the 'fittest' from one population are chosen and then mutated further.

You can see that even though this method can guarantee an exhaustive search in the hypothesis space and might even come up with unexpected and novel results to any solution. But you might also have noticed the huge drawback - it all might take quite some time. The most popular systems using this approach is *ADATE* [4]. It is a very exciting and powerful representative of *evolutionary* programming and has had some amazing success in the modification and improvement of algorithms like image segmentation. But as we are trying to fit machine learning and software engineering together on an everyday-application-base it seems inadvisable to use an approach which might force the programmer to wait infinite time for a solution to his problem. The point is that *ADATE* is never really finished - you never know if there will be an improved solution next to the current best.

There has been an approach to combine functional and evolutionary techniques ([10]) in order to tackle this problem - but for now we must conclude that it is of no relevance to us for the current project and we will go with *Inductive Functional Programming* and *Igor*.

[2]see `http://www.rulequest.com/Personal/`

[3]see `http://www.doc.ic.ac.uk/~shm/`

[4]see `http://www-ia.hiof.no/~rolando/`

3.3 Igor

Summers' theories have been taken up again in [11], 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 [7] 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.* [5]

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]) = 1
length([y,x]) = 2
length([y,x,z]) = 3
```

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)))
```

Essentially, this is what we will be concerned with later on as we are going to identify some possible applications of this within an object-oriented software engineering project. Special caution has to be taken since this system is very obviously purely functional so we are going to be concerned with finding out how to bring the two concepts together in order to gain something productive and maybe even something useful in the course.

For this, the next step is to prepare some insight into the case study along with a short analysis of the outcoming implementations in regard to usefulness for our purposes to help a programmer along the way by having *Igor* eliminate a couple of tasks from his/her to do list.

[5] see `http://maude.cs.uiuc.edu/`

4 Case Study: Software Engineering

The case study we picked for our attempt to find ways to automate parts of a software engineering process with the help of *IP*, were the assignments from the software engineering course held at Bamberg University in 2008 [6].

4.1 Assignment Outline

A set of project groups were given the task to design and implement a 'Thesis Store', an administrative tutoring system for the departments staff. Mandatory components were administration of potential thesis topics together with a functionality to assign a concrete topic to a student and a tutor along with a scheduling module, which would help student and tutor keeping track of the thesis' milestones and deadlines. Data was to be persisted with the help of a database and the preferred programming language was Java. The project itself had to be engineered along the *Presentation-Control-Mediator-Entity-Foundation (PCMEF)* architecture as laid out in [1].

Since only two groups came up with a solution which was comprehensive and correct enough to make use of it for our purpose we will only be concerned with their results. As those projects still suffered as much from the limitations of capacities as those of the other groups they are still not complete in the sense of a proper software engineering project. However, they provide a good enough insight in the outline and the fundamental structure of what bigger-scale productive projects might look like. As the functionality of the system is understandably easy we hope to find some useful parts to instrumentalise for our purposes.

But before we dig deeper, let us first have a look at a rough overview over the implementation details of both groups.

	Methods	void out	simple out	complex out	void in	simple in	complex in	recursion
TOTAL	634	260	161	210	292	184	160	88
%		41.01	25.39	33.12	46.06	29.02	25.24	13.88

Table 1: Group 1 Statistics

	Methods	void out	simple out	complex out	void in	simple in	complex in	recursion
TOTAL	1253	659	222	370	623	188	441	101
%		52.59	17.72	29.53	49.72	15	35.2	8.06

Table 2: Group 2 Statistics

In order to understand the single column data the concepts have to be defined like this:

- *void*: empty construct
- *simple*: simple datatype like **string** or **int**
- *complex*: anything else than void or simple (-> objects)

[6]Dept. of Practical Computer Science at the University of Bamberg `http://www.uni-bamberg.de/pi/`

The most obvious reason for the high percentage of void input/output parameters can be found in the fact that the intensive usage of entities resulted in a high number of getter/setter who are responsible for many of those void methods. From these numbers we can already guess that the first group must have done a much more thorough job in this aspect since there are twice as many functions used than in the second group's implementation.

But these numbers are of no major importance to us right now, since we are interested in functionality and here we are stricken by the fact that the number of recursive/iterative statements is considerably low. In addition, a closer look at the implementation showed that the majority of those recursive statements are iterations over a collection. The reasons for this can for one be found in the not very complex nature of the assignment. Also the usage of a database does rule out many interesting tasks such as sorting, combining and searching within data. Since the process of software engineering per se produces a lot of overhead there are of course many lines of code which don't have anything to do with the actual processes the final system is supposed to be executing. Many of the layers used to abstract the software in order to meet the identified business process are just 'containers' if you will. The level of encapsulation is very consequently increased along the way of building the software, which makes sense in terms of reusability and maintainability. But since we are interested in the *core* functionality here we have to dig a little deeper and peel of all the layers around the system's core.

Finally there was one suitable example (see listing 30) we might be able to pull something out of in order to have *Igor* find a solution to this problem. The obvious problems these methods suffer from are that in case of the *compareStrings* method the functionality of Java is properly used by simply applying the *contains(Object o)* method on **String**. The method *compareStringArrays* is a little more interesting. Encapsulated in two methods we find a nested loop within a loop - is this the point where we might try and find a recursive solution which is a little more elegant and maybe even faster?

Let us just say for now that there is a strong feeling of uncertainty if there can be a proper use of automated program induction within such a project for all the reasons just mentioned. But since this project is certainly not nearly as complex as a real one would be we can still not finally give up on this subject - we just cannot say that for sure right now. What we can do is pretend that it *might* still be nice to have some support of this kind available in any programming task. Remember, we already had to take *RSA* out of the equation due to lack of access to the code building process. So let us presume for now that we are trying to help a programmer in *any* situation to have an *IP* system to help him find solutions to a problem, later on we can and will be worried about the problem putting this back into the context of software engineering.

5 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 synthesised 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 datatypes 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 follows the pattern-matching concept, in which rules are written in *PROLOG*:

$Result(EquationRHS) \leftarrow Rule/Pattern(EquationLHS)$

5.1 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 **variables** 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 blackboxes, 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 errorhandling is not the major part of our concern, let it just be introduced as blackbox - 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 constructors (three blanks → 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.

5.2 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 blackbox 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].
```

5.3 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 5.1 and an object trying to call a method by an identifier. The idea is to get a matching process like:

$Identifier \times MethodList \rightarrow Method$

This would be quite straightforward since we already know that any object contains a list of methods which all carry an identifier. So now it would just be a simple matter of going through the list and returning the one method matching the identifier provided. Of course if we would like to be pedantic we would have to ensure that no two methods can have the same identifier unless they differ at least in return type or number/datatype of requested argument/s.

Now let us try all this on *Igor* for a change and see if there is any chance that everything we have conceptualised so far can be processed by the system. If it should fail on these simple concepts there is no need to go on from here. For this we define a specification like in listing 20.

Listing 5: Identifier Match

```
sorts InVec 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 *NullPointerException* to be thrown in case an identifier is not found within the method list. The exception is here derived from *Method* which is not quite clean but since this is only used for testing it is not going to be a problem.

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: 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 .
```

Listing Info: [] is the empty list, *cons* the constructor for lists as known from *LISP*, *mm* a constructor for *Method*, *exc* a constructor of the type *NPException* (NullPointerException)

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: 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. 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 20) 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.

Info: $X1$ and $X2$ are identifiers, $X3$ is a list, *dt* a datatype and *parlist* a list of parameters

1. $exc \leftarrow match(X1, [])$
2. $X1 \leftarrow Sub1(X1, cons(mm(X2, dt, parlist), X3)) \wedge \neg(X1 == X2)$
3. $X3 \leftarrow Sub2(X1, cons(mm(X2, dt, parlist), X3)) \wedge \neg(X1 == X2)$
4. $match(Sub1(X1, cons(mm(X2, dt, parlist), Sub2(X1, cons(mm(X2, dt, parlist), X3))))) \leftarrow match(X1, cons(mm(X2, dt, parlist), X3)) \wedge \neg(X1 == X2)$
5. $mm(X1, dt, parlist) \leftarrow match(X1, cons(mm(X2, dt, parlist), X3)) \wedge X1 == X2$

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.

5.4 Concerning Variables

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 - so there is no sense in repeating the procedure all over.

5.5 Messages

As already mentioned, we are going to relate every action within our system to messages. In 5.1 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"] .
```

Listing Info: *s* is the successor-operator on natural numbers, *t* and *f* the boolean constants, *cpar* a constructor like *cons* on a list of parameters, <> is the empty parameter list, *msg* a constructor of the type *Message*

Next to the already known definitions of *message* and the usual list operations there are some more definitions. Since we have to provide natural numbers as *peano numbers* to *Igor*, there has to be a *successor* operator (s), as well as we need the boolean values *true* and *false*. 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:

Info: $X1$ is a natural number

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

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. $X1 \leftarrow Sub19(s(s(X1)))$
2. $t \leftarrow method(0)$
3. $f \leftarrow method(s(0))$
4. $method(Sub19(s(s(X1)))) \leftarrow method(s(s(X1)))$

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.

5.6 Back Into Perspective

As we have come to realise so far, it is possible to express some basic object oriented concepts functionally using touples as datastructure. We have seen that *Igor* can even learn some basic notions like referencing methods within objects (5.3) or using a simple protocol like the one we called *message-passing* (5.5). Furthermore it has been demonstrated that objects, methods and variables can be specified in a simple functional way. It is about time to try and bring it all together and find out how we can use all this in a practical way for our eclipse plug-in. Before we go ahead and do this, let us just come back to our *object* specification (see listing 18). As you can tell from the example we are trying to round up what we have done so far by putting it all together in one example. What we now do is apply the method-call on an entire object, not just on the list of methods. We just add one more layer to it - the result is still the same. This means that we could now go ahead and take this approach all the way until we have successfully modelled a complete object oriented world. Since this is not the main focus of this thesis we are going to leave it at that point and get back to our primary objective which is trying to find out how programmers can benefit from this.

As we have come to see, our case study did not provide a great variety of examples which we could use for our purpose. But while we are still at it, why don't we just pick one of the things we came across in large numbers like iterating over a collection? It is true that there is not a high relevance to it since Java offers a great deal of functionality to do it very quickly. On the other hand we have just found out that *Igor* does not seem to struggle too much with the basic object oriented concepts, so why don't we just try and find out if it can handle something a little more close to real programming than passing messages and resolving method identifiers?

In listing 24 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.

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) ,{})) .
[...]
```

Listing Info: *put* and *put*2 are two list constructors like *cons*

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*. 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 26 we go the same way we already did with methods and objects. Another layer of abstraction is added or, if you will, some more object oriented 'overhead' by adding more detail into the method call itself. 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 27) 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.

So far it should have become evident that it is possible to formalise a simple object oriented system in *Maude* and have *Igor* synthesise program fragments in this paradigm.

But for now we only have been playing around with very basic examples and we still have not found an answer to the question if and how a programmer might benefit from this. In the next chapter we will be concerned with *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 the system and thus getting help by an automated program induction system.

6 Plug-in for Eclipse: autoJava

6.1 Requirements & Installation

This plug-in was built with *Eclipse 3.4.1 (Ganymede)* on an AMD X2 4200 with 2MB RAM, running on Kubuntu 8.10.

In order to get the plug-in running on your system you have to make sure you have the following installed on your system:

- an up-to-date Linux distribution (Debian, Ubuntu, Gentoo, SuSe, ...)
- the Eclipse development platform
- the latest *Maude* binaries (download from here: `http://maude.cs.uiuc.edu/`)
- *Igor* (download from here: `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`) version 2.2 or 2.3
- the file 'spec_frame.maude' (in the plug-in's 'res' folder) somewhere on your system

When you have installed the plug-in by simply copying the *autoJava.jar* into your Eclipse plug-in folder you can go ahead and run Eclipse.

6.1.1 Some Notes

This plug-in is, as *Igor*, an experimental prototype and not meant for industrial use in any kind. Do not expect a bug-free software either, just use it in order to get accustomed with the system and explore the possibilities it may or may not hold.

6.2 autoJava in Theory

On the whole the plug-in consists of three major components:

1. antlworks & annparser (6.2.1)
2. easyigor (6.2.2)
3. autojava (6.2.3)

While 1. is concerned with parsing the syntax for the specification as well as building the structure of simplified parenthesised expressions, 2. is the 'application layer' (to use software engineering terminology) and 3. is just the 'presentation layer' which integrates the application as a plug-in into the Eclipse workbench. The basic principle here is that we have the user annotate the methods he wants to be automatically induced by *Igor*. For this a simple annotation specification has been designed which is going to be described in 6.2.1. Before going into detail we are going to run through the program's routine like depicted in figure 1.

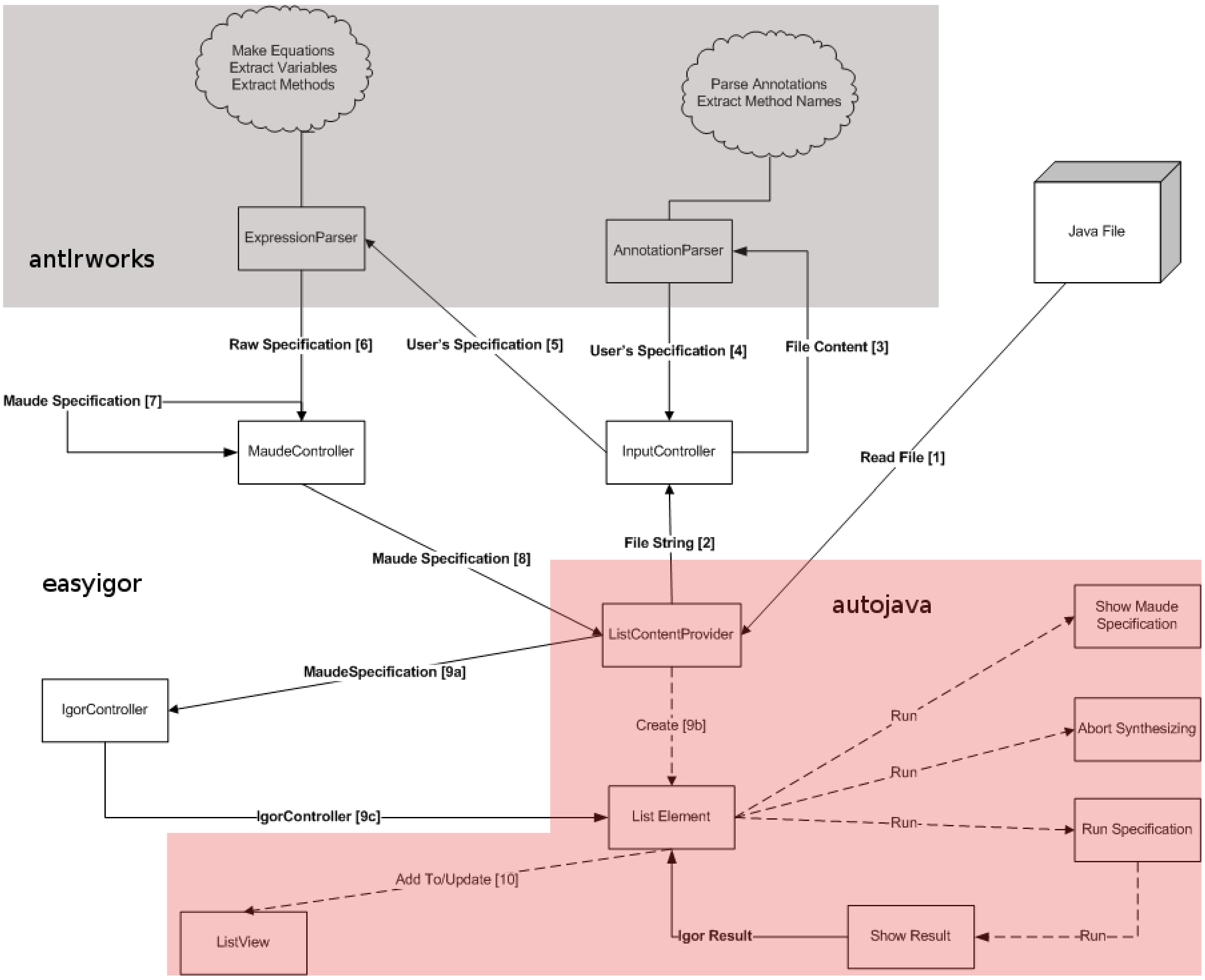


Figure 1: autoJava Architecture

1. **ListContentProvider** reads the currently active java file
2. send the file content as string to the **InputController**
3. pass it onto the **AnnotationsParser** which extracts the annotations containing a specification and validates syntax
4. specification is returned to **InputController** as **SimpleIgorAnnotationPair** for every method found within the java string
5. results are passed on to the **ExpressionParser** which generates the equations and extracts variables and methods from them
6. the raw specification is forwarded to the **MaudeController**
7. **MaudeController** takes the template *Maude* file and inserts the raw specification data
8. *Maude* specification returned to **ListContentProvider**

9. a new list element is created for the specification (a), the *Maude* specification is used to create an **IgorController** (b) which is stored within the list element (c)

The procedures 3 - 9 are repeated for every specification provided by an annotated method within the Java file. At the end the **ListView** contains elements which each contain an **IgorController** to be used to trigger the program synthesis, show the specification or insert the result into the method's body inside the java document.

6.2.1 Antlrworks

Most important for this component are the parsers for annotations (*AnnotationsParser*) and for the expressions (*ExpressionsParser*). The former is used to extract annotations from the source code, find all the methods contained in a java file and, finally, check the syntax of the annotations. The latter is used to build an abstract syntax tree of the expressions in order to transfer their structure into valid *Maude* syntax. They have been built by the *IgorAnnotation* (listing 32) and the *ParExpression* grammar (listing 31). In the *ExpressionParser* on the one hand, user provided equations are parsed and with the help of an abstract syntax tree their features are translated into a valid *Maude* equation. If the equation should contain a list, it will be converted from a more object oriented notation into the functional equivalent like this:

$[a, b, c] \rightarrow cons(a, cons(b, cons(c, [])))$

Lists can be nested or contain method calls, all this is extracted along the way of building the syntax tree. This information will be used by the time the *Maude* specification is built, as variables have to be declared before they can be used.

Secondly, the *AnnotationsParser* is used in order to validate the syntax of the annotated *Igor* specification. A valid example would look like illustrated in listing 14

Listing 14: Correct Annotated Specification

```
/**
*@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(){


}
```

As we can see from the example, the specification consists of three parts:

- Meta-Part (@IgorMETA)
- Equations (@IgorEQ)
- Name of annotated method (@Method)

Those three have to be specified in the following manner:

$@[identifier]([content]);$

Meta content needs three parameters like in the example.

- methodName: the name for the induced method in the *Maude* specification (can differ from the name of the java method)
- retValue: the datatype of the value to return by the induced method
- params: arguments for the function - if there are more than one they need to be put one after another with blanks between them ($params =$ '$arg1\ arg2\ arg3...$'.)

Those parameters have to be annotated as key value pairs like this:

$paramName =$ '$value$'.

Finally the equations need a little more time to explain since this is where the knowledge is put in. In the most basic form they need to be as follows:

$(lhs) = rhs.$ as single equations, wrapped by:

$equations = \{[content]\}$

An equation's left-hand-side must be put in parentheses to allow the parser to recognise multi-component input like this:

$([a, b], [c, d]) = [a, b, c, d].$

So on the left hand side the input can be specified in the following way:

$(arg1, arg2, arg3, ...)$

Since the Antlr grammar files are practically *BNF*, all this can be found summarised in listing 32.

6.2.2 Easyigor

The application-layer is basically an independent component which could be used for any other purpose. It uses the parsers and in essence controls the whole process of building a *Maude* specification. Most important parts are:

- **InputController**: process user input (mainly extract specification, parse syntax...)
- **MaudeController**: create *Maude* specification with all the parts handed over from the **InputController**
- **IgorController**: serves as a 'remote control' for *Igor*, allowing to start, stop and read results from it (makes use of the java-wrapper *App2IgorInterop* [7])

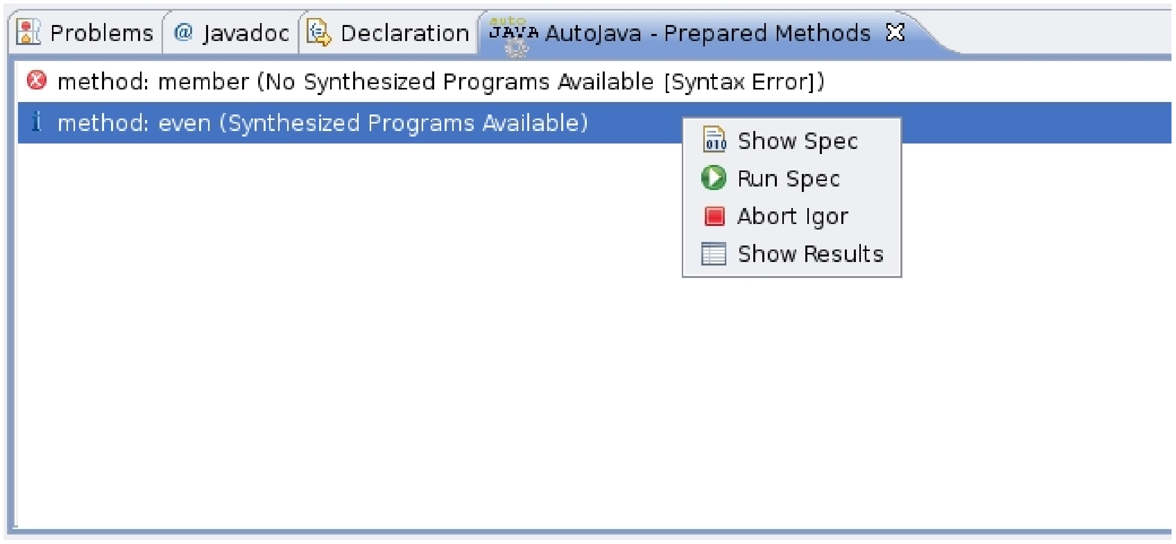
[7]see `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`

- **AppConfiguration**: the memory and central point of the application (provide access to controllers, instantiate the logger)

So the major task of this layer is to provide the interfaces implemented by the controllers to a presentation-layer which in this case is *autoJava*.

6.2.3 Autojava

Autojava integrates the application-layer into the Eclipse workbench, wrapping it in a plug-in. The user interface is contained in a list view which displays the methods found in the active java file (figure 2)



method: member (No Synthesized Programs Available [Syntax Error])

method: even (Synthesized Programs Available)

Figure 2: autoJava List View with Context Menu

Figure 3: Syntax Error

Figure 4: Syntax OK

In this view, information about the status of the method is displayed. Here we can see that there are two methods (*member* and *even*) and one of them apparently contains a syntax error. The validity of syntax is for one displayed textually and, in addition, with the help of an icon (figures 3 and 4).

The control over *Igor* can be triggered with the context menu on every list item (figure 2). As the options are self-explanatory there is nothing to go into detail here, just know that if you select *Show Spec*, a message window will pop up showing the specification, while *Show Results* will insert the synthesised program into the body of the java method. As soon as you start *Igor* the list item will keep you updated about the progress, as soon it has finished it will display 'DONE', which is the point from which on you can access the results and put them into them method body.

6.3 autoJava in Practise

To start off we go ahead and just use the example specification used earlier to introduce the annotation language in listing 14. Supposedly, this specification will result in a program which will automate the last element in a list. After creating an empty method called ‘last‘ in a java file, the specification is added to the method’s annotation. If we now run the specification and have the results inserted into the java file it should look like listing 15.

Listing 15: 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] .)

}
```

You can see that it looks like this problem has been solved correctly but you will notice that the result comes in *Maude* syntax. As the focus of this thesis was to generally explore if and how an integration of *Igor* into a development environment was possible and if the object oriented paradigm was principally possible, the attempt to translate the results into java would have simply put the whole out of proportion. But it should be mentioned that a parser for those results already exists and so a translation is basically possible. Also, the implementation lacks the object oriented flavor we have already demonstrated on *Igor*. But you will have come to see in chapter 5 it is not a great difference for *Igor* how you wrap things and so it should become clear that together with a general translation of the results into java syntax this is a major *to do* for a next version of this plug-in, but one which is essentially ready to be done. We will come back to this later when we draw a resume, so let us for now stick with what we have and go ahead and find some more challenging examples.

As mentioned earlier we found a lot of procedures on collections in our case study and we have already successfully tested a specification on *Igor*. So how about trying to do the same again without having to generate the whole specification ourselves by using *autoJava*. The specification to be used should be familiar already, but of course this looks a little different in our simple notation:

Listing 16: Iteration over a collection

```
/**
* @IgorMETA(
*    methodName   = "iterate".
*   retValue     = "List".
*   params       = "List".
* );
* @IgorEQ(
*    equations = {
*    "([]) =[]".
*    "([x])=[met(x)]".
*    "([x,y])=[met(x),met(y)]".
*          "([x,y,z])=[met(x),met(y),met(z)]".
*   }
* );
* @Method(iterate);
*/
```

The resulting equations are once again altered in order to be more readable:

Info: $X1$ is an Object, $X2$ a List

1. $met(X1) \leftarrow Sub1(X1, X2)$
2. $iterate(Sub5(cons(X1, X2))) \leftarrow Sub2(cons(X1, X2))$
3. $X2 \leftarrow Sub5(cons(X1, X2))$
4. $[] \leftarrow iterate([])$
5. $cons(Sub1(cons(X1, X2)), Sub2(cons(XX2)) \leftarrow iterate(cons(X1, X2)$

Comparing these results with the ones from the example we have already run through *Igor* (see listings 24 and 25) you will realise that they correspond. As we surely agreed before, there is actually no need for a programmer to give something away to program induction when he can have it solved with the help of the java libraries just as easily (foreach-loop). But by now we have already proved, that a concept which is frequently used in object orientation can be transferred into the functional world and synthesised by *Igor*.

Another very basic kind of operation in object oriented programming is string manipulation. We can very elegantly fit this into our current perspective since a string is nothing less than a list of characters. As we have already seen, lists are the strength of *Igor* and so there will be one last example before we will begin to summarise the findings of the last chapters.

Imagine, we want to transform a given string into all lowercase letters. Of course you will again say 'it's been done', but for now it is important to understand that this is supposed to be foundations of object orientation in inductive program synthesis. So for our purpose it shall be a complex enough task to hand over to *Igor*. Before we start we have to come up with a solution to the problem that we are not going to use concrete string values in our example equations. This means that we have to find a way expressing the caption of a letter. As we found out earlier, we can use any kind of 'dummy' method for our purpose, so let us quickly introduce two of them:

- *uc() - upper case*
- *lc() - lower case*

By using them as attributes for a string we can now go ahead and produce our specification like the one in listing 17.

Listing 17: To-Lower Specification

```
/**
*@IgorMETA(
*   methodName   = "tolower".
*   retValue     = "String".
*   params       = "String".
*);
*@IgorEQ(
*   equations = {
*       "([]) =[]".
*       "(uc(a))=lc(a)".
*       "(lc(a))=lc(a)".
*       "([uc(a)])=[lc(a)]".
*       "([lc(a)])=[lc(a)]".
*       "([uc(a),uc(b)])=[lc(a),lc(b)]".
*       "([lc(a),uc(b)])=[lc(a),lc(b)]".
*       "([uc(a),lc(b)])=[lc(a),lc(b)]".
*       "([lc(a),lc(b)])=[lc(a),lc(b)]".
*
*   }
*);
*@Method(tolower);
*/
```

This is already enough to have *Igor* compute one result (see listing 28), which correctly iterates through the string, gradually transferring every capital letter to lowercase. We could now go on trying to find more examples of fundamental procedures on objects within the *OO* paradigm. You will have realised that the special strength of *Igor* concerning lists and recursion had to be used as a major component when bringing the two different worlds together. At the same time, it should have become evident that there is a way to unite them like we were trying to do in chapter 5. In this chapter, the main focus was to try and find a practical approach to this whole problem and the most important notion at the end of it must be that there seems to be a possibility of bringing all this together on a larger scale, but the restrictions on time and workload did not allow for it to happen in this thesis.

6.4 autoJava in Conclusion

There are of course some shortcomings of the implementation since the primary focus of this thesis was to find a way to bring object orientation and *Igor* together. The plug-in itself was then created with the objective to simplify the way a specification has to be produced for *Igor*. Some late changes to implement method calls seem to have had a negative effect on the syntax checking, which is now not as restrictive as it initially was. It is one of the major drawbacks of the current version, as is the fact that since the main focus was on the processing of lists there may now be some problems which cannot be constructed with this new annotation convention.

Some more known errors are listed below:

- Generation of specification always on the whole file - this causes slowdowns.

  FIX: only generate those specifications which have changed or are new.

- Numbers within identifiers are basically possible, but practically not since every number in the equations is replaced by its Peano value. Also Peano numbers are only generated from 0 - 9, two digit numbers fail and mess up the specification

  FIX: use a more intelligent algorithm to extract numbers from the equations.

- Non related javadoc comments may make it impossible to detect the annotations.

  FIX: find a more sophisticated way to tell javadoc and *Igor* annotations apart.

- The change listener on the editor workbench is sometimes not correctly attached. The file has to be closed and reopened to get it working again.

  FIX: some more time spent on the listener and the list view should solve this problem.

As pointed out before, autoJava is a prototype and therefore there is no claim that it is complete or fully functional. The objective of simplifying the way to provide specifications has been solved partially, when it comes to list functions it should be complete. This is indeed the most important thing since it has been pointed out that the way to realise object orientation in *Igor* uses lists on a large scale.

7 Watch what we did

Finally, it is time to draw a conclusion and come up with a judgement on all the aspects covered by now. In chapter 1 the focus of the thesis was described as bringing ‘object oriented and functional programming together‘ and trying to put this into the context of software engineering. This was the first problem to arise since neither *RSA* could be used in the way intended, nor could we make proper use of the case study (see chapter 4).

This case study suffered from the problem that the main focus was not on functionality, but on the application of software engineering techniques which left the remaining project with very little functionality other than the many layers of the *PCMEF* architecture. The only useful code sample identified (listing 30) was a glimpse of light but it turned out to be very difficult to use. One main problem is that the algorithm is slightly awkward, it doesn't allow to specify a proper base case for *Igor*, which is why it took very long to realise that this task could not be tackled very easily. In the end it was abandoned for the sake of progressing with the more relevant work, but it would still be very interesting to try and tackle this problem again. This is something which has to be left open and illustrates at the same time one of the main issues around this thesis.

In chapter 5.1 the main understanding was to ‘start small‘ and this is what can be said for this entire piece work. Along the way were many small problems that did not matter so much at first glance. The case study is one of those, as was the generation of the syntax grammars for the specification annotation, which is still not entirely correct. There were, on the other hand, a number of things which were quite straightforward and surprisingly intuitive. The whole procedure of modelling and testing the object oriented functionality within *Igor* was still very time demanding but the final feeling is that there were some concrete results as chapter 5 shows.

We have asked the question, whether this could be a benefit for a programmer at work. In order to find this out the steps to be taken were:

1. create an interface to interact with *Igor*
2. keep the specification simple and closely connected to object orientation
3. be able to tackle object oriented problems
4. produce a usable program output

Considering those aspects, it can be said that 1. has been fulfilled simply by providing the plug-in as it is. The second point can be checked as well with the minor comment on the issues connected with the specification language in general. Even though it (for now) was not possible to solve one of the problems out of the case study we were successfull in other procedures like processing collections or strings. So there must be said that this aspect is partially fulfilled even though it clearly has to be elaborated on. Finally the synthesised program has not been processed in the current version and is still not usable within any object oriented programming language. But since *Igor's* results are ready to be parsed and translated into other languages (XSL for example in [6] and [5]) we can agree that this is a minor issue which would be quite tedious to carry out but manageable nonetheless. Even more importantly the way to implement the specification annotation provides a very elegant way to interconnect with *RSA*. Since it allows the programmer to annotate any method as early as the modelling takes place it seems not too far from reality to have him do this not in natural language but with a specification for *Igor*. Since those annotations are translated into javadoc comments we would have successfully imported our specification into the generated code. Here the programmer can elaborate on this using autoJava and finally have some methods generated automatically.

It should be clear that we have not been able to come up with a running program that seamlessly integrates within *RSA* and generates executable java code. But we have proven that *Igor* is not only able to handle object oriented procedures, it can also synthesise them as illustrated in chapter 5. The next steps to be taken are to create a complete model of an object oriented system and integrate this via background knowledge into our running plug-in. This becomes possible since we have used a concrete template which can of course be enriched by addidtional modules, variables or equations. Since these would be a knowledge base for object oriented processes in general there is no need to have the programmer specify them every time. By employing this text template the door is open for any kind of alteration - even on runtime.

All in all the direction is set as there exists a common foundation between functional programming and object orientation. After having considered a number of facts and examples the resume of this thesis would be that it is possible to use methods of program synthesis in software engineering. Further, a way has been introduced to take down the ‘barrier‘ between functional and object oriented programming as a starter for future development. The plug-in developed to demonstrate some basic functionality and how it might be possible to integrate into tools used in software engineering (Eclipse) has clearly demonstrated that the user interface is ready to support *Example-Driven Programming* in this field and so it will be quite thrilling to see what can be made out of this.

References

[1] Leszek Maciaszek; Bruc Lee Liong; Stephen Bills. *Practical Software Engineering.* Pearson/ Addison Wesley, 2005.

[2] Ruknet Cezzar. *A Guide to Programming Languages.* Artech House, 1995.

[3] Allen Cypher. Programming repetitive tasks by demonstration. In *Watch What I Do: Programming by Demonstration*, pages 205–217. The MIT Press, 1993.

[4] Anthony J. Field; Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[5] Thomas Hieber. Transportation of the JEdit plug-in ProXSLbE to eclipse. Technical report, Otto Friedrich University of Bamberg, 2008.

[6] Martin Hofmann. Automated construction of xsl-templates: An inductive programming approach. Master's thesis, Otto Friedrich University of Bamberg, 2007.

[7] 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.

[8] Thomas Kühne. *A Functional Pattern System for Object-Oriented Design.* PhD thesis, Darmstadt University of Technology, 1999.

[9] H. Lieberman. Tinker: A programming by demonstration system for beginning programmers. In *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.

[10] Neil Crossley; Emmanuel Kitzelmann; Martin Hofmann; Ute Schmid. Combining analytical and evolutionary inductive programming. In *Proceedings of the Second Conference on Artificial General Intelligence.* Pascal Hitzler; Marcus Hutter.

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

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

8 Appendix

8.1 Maude Codesamples

Listing 18: 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 19: Object Result

```
reduce in IGOR : generalize('OBJECT) .
rewrites: 256100 in 564ms cpu (562ms real) (454049 rewrites/second)
result HypoList: hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['?X0:Identifier,
    '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'X4:Identifier if '_==_[
    'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
    'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
    'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X2:Identifier,
    'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['?X0:Identifier,
    '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'id1.Identifier if
    '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
    'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
    'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X2:Identifier,
    'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X1:Identifier,
    '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'X4:Identifier if '_==_[
    'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
    'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
    'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X2:Identifier,
    'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X1:Identifier,
    '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'id1.Identifier if
    '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
    'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
    'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X2:Identifier,
    'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X2:Identifier,
    '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'X4:Identifier if '_==_[
    'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
    'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
    'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X2:Identifier,
    'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)
```

```

nextHypo

hypo(true, 3, eq 'mcall['__['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [
      none] .
ceq 'Sub1['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = '__['X2:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'id1.Identifier if
     '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'mcall['Sub1['__[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:
           Identifier],'Sub2['__['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:
           Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'met['X2:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['__['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [
      none] .
ceq 'Sub1['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = '__['X4:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'X4:Identifier if '_==_[
     'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'mcall['Sub1['__[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:
           Identifier],'Sub2['__['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:
           Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'met['X2:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['__['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [
      none] .
ceq 'Sub1['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = '__['X4:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'id1.Identifier if
     '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'mcall['Sub1['__[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:
           Identifier],'Sub2['__['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:
           Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'met['X2:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['__['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [
      none] .
ceq 'Sub1['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = '__['?X0:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'X4:Identifier if '_==_[
     'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'mcall['Sub1['__[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:
           Identifier],'Sub2['__['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:
           Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['__['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [
      none] .
ceq 'Sub1['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = '__['?X0:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'id1.Identifier if
     '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'mcall['Sub1['__[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:
           Identifier],'Sub2['__['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:
           Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['__['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
      List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

```

```
nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X1:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'X4:Identifier if '_==_[
     'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X1:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'id1.Identifier if
     '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X2:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'X4:Identifier if '_==_[
     'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X2:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'id1.Identifier if
     '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3, eq 'mcall['___['X1:Identifier,'`[`].List,'`[`].List],'X2:Identifier] = 'exc.NPException [none] .
ceq 'Sub1['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = '___['X4:Identifier,
     '`[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'X4:Identifier if '_==_[
     'X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'mcall['Sub1['___[
     'X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier],'Sub2['___['X1:Identifier,'`[`].List,
     'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,'`[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier] = 'met['X4:Identifier,
     'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)

nextHypo
```

```

hypo(true, 3, eq 'mcall['___['X1:Identifier,''[`].List,''[`].List],'X2:Identifier] = 'exc.NPException [
    none] .
ceq 'Sub1['___['X1:Identifier,''[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
    List]],'X4:Identifier] = '___['X4:Identifier,
    ''[`].List,'X3:List] if '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'Sub2['___['X1:Identifier,''[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
    List]],'X4:Identifier] = 'id1.Identifier if
    '_==_['X2:Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,''[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
    List]],'X4:Identifier] = 'mcall['Sub1['___[
    'X1:Identifier,''[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:
        Identifier],'Sub2['___['X1:Identifier,''[`].List,
    'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]],'X4:Identifier]] if '_==_['X2:
        Identifier,'X4:Identifier] = 'false.Bool [none] .
ceq 'mcall['___['X1:Identifier,''[`].List,'cons['met['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:
    List]],'X4:Identifier] = 'met['X4:Identifier,
    'dt.DType,'parlist.ParamList] if '_==_['X2:Identifier,'X4:Identifier] = 'true.Bool [none] .)
```

Listing 20: 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 21: Identifier-Match Result

```
reduce in IGOR : generalize('IDENTIFIER-MATCH) .
rewrites: 208528 in 440ms cpu (438ms real) (473898 rewrites/second)
result HypoList:

hypo(true, 3,
eq 'match['X1:Identifier,'`[`].List] = 'exc.NPException [none] .

ceq 'Sub1['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] = 'X1:
    Identifier if
    '_==_['X1:Identifier,'X2:Identifier] =
    'false.Bool [none] .

ceq 'Sub2['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'X3:List if '_==_['X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'match['Sub1['X1:Identifier,'cons['mm['X2:Identifier,
    'dt.DType,'parlist.ParamList],'X3:List]],'Sub2['X1:Identifier,'cons['mm['X2:Identifier,
    'dt.DType,'parlist.ParamList],'X3:List]]] if '_==_[
    'X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
  'mm['X1:Identifier,'dt.DType,'parlist.ParamList] if
     '_==_['X1:Identifier,'X2:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3,
eq 'match['X1:Identifier,'`[`].List] = 'exc.NPException [none] .

ceq 'Sub1['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
     'id1.Identifier if '_==_['X1:Identifier,'X2:Identifier] =
    'false.Bool [none] .

ceq 'Sub2['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] = 'X3:List
    if '_==_['X1:Identifier,'X2:Identifier] =
    'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'match['Sub1['X1:Identifier,'cons['mm['X2:Identifier,
    'dt.DType,'parlist.ParamList],'X3:List]],'Sub2['X1:Identifier,
    'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]]] if
    '_==_['X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'mm['X1:Identifier,'dt.DType,'parlist.ParamList] if
    '_==_['X1:Identifier,'X2:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3,
eq 'match['X1:Identifier,'`[`].List] = 'exc.NPException [none] .

ceq 'Sub1['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'X1:Identifier if '_==_['X1:Identifier,'X2:Identifier] =
    'false.Bool [none] .

ceq 'Sub2['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'X3:List if '_==_['X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'match['Sub1['X1:Identifier,'cons['mm['X2:Identifier,
    'dt.DType,'parlist.ParamList],'X3:List]],'Sub2['X1:Identifier,
    'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]]] if
    '_==_['X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'mm['X2:Identifier,'dt.DType,'parlist.ParamList] if
    '_==_['X1:Identifier,'X2:Identifier] = 'true.Bool [none] .)

nextHypo

hypo(true, 3,
eq 'match['X1:Identifier,'`[`].List] = 'exc.NPException [none] .

ceq 'Sub1['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'id1.Identifier if '_==_['X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'Sub2['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] = 'X3:List
    if '_==_['X1:Identifier,'X2:Identifier] =
    'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'match['Sub1['X1:Identifier,'cons['mm['X2:Identifier,
    'dt.DType,'parlist.ParamList],'X3:List]],'Sub2['X1:Identifier,
    'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]]] if
    '_==_['X1:Identifier,'X2:Identifier] = 'false.Bool [none] .

ceq 'match['X1:Identifier,'cons['mm['X2:Identifier,'dt.DType,'parlist.ParamList],'X3:List]] =
    'mm['X2:Identifier,'dt.DType,'parlist.ParamList] if
    '_==_['X1:Identifier,'X2:Identifier] = 'true.Bool [none] .)
```

Listing 22: 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 23: OO-Call Result

```
reduce in IGOR : generalize('OO−CALL) .
rewrites: 230564 in 448ms cpu (449ms real) (514619 rewrites/second)
result Hypo:
hypo(true, 3, eq 'Sub19['msg['cpar['s['s['X1:Nat]],'<>.ParamList],'null.Object]] =
  'msg['cpar['X1:Nat,'<>.ParamList],'null.Object] [none]
     .
eq 'method['msg['cpar['0.Nat,'<>.ParamList],'null.Object]] = 'msg['<>.ParamList,'t.Bool] [none] .

eq 'method['msg['cpar['s['0.Nat],'<>.ParamList],'null.Object]] = 'msg['<>.ParamList,'f.Bool] [none] .

eq 'method['msg['cpar['s['s['X1:Nat]],'<>.ParamList],'null.Object]] =
  'method['Sub19['msg['cpar['s['s['X1:Nat]],'<>.ParamList],'null.Object]]] [none]
```

Listing 24: 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 25: Iterate-Collection Result

```
reduce in IGOR : generalize('ITERATE_COLLECTION) .
rewrites: 21887 in 52ms cpu (50ms real) (420879 rewrites/second)
result Hypo:
hypo(true, 2, eq 'Sub1['put['X1:Object,'X2:Collection]] = 'met['X1:Object] [none] .

eq 'Sub2['put['X1:Object,'X2:Collection]] =
  'iterate['Sub5['put['X1:Object,'X2:Collection]]] [none] .

eq 'Sub5['put['X1:Object,'X2:Collection]] = 'X2:Collection [none] .

eq 'iterate['`[`].Collection] = '`{`}.ResultCollection [none] .

eq 'iterate['put['X1:Object,'X2:Collection]] =
  'put2['Sub1['put['X1:Object,'X2:Collection]],'Sub2['put['X1:Object,'X2:Collection]]] [none] .)
```

Listing 26: 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
```

Listing 27: Foreach-Do Result

```
reduce in IGOR : generalize('FOREACH-DO) .
rewrites: 29271 in 64ms cpu (62ms real) (457330 rewrites/second)
result Hypo:
hypo(true, 2, eq 'Sub1['push['X1:Object,'X2:Collection]] =
  'call['X1:Object,'id1.Identifier,'dt1.DType,'pp.ParamList] [none] .

eq 'Sub2['push['X1:Object,'X2:Collection]] =
  'it_apply['Sub5['push['X1:Object,'X2:Collection]]] [none] .

eq 'Sub5['push['X1:Object,'X2:Collection]] =
  'X2:Collection [none] .

eq 'it_apply['`[`].Collection] = '`[`].Collection [none] .

eq 'it_apply['push['X1:Object,'X2:Collection]] =
  'push['Sub1['push['X1:Object,'X2:Collection]],'Sub2['push['X1:Object,'X2:Collection]]] [none] .)
```

Listing 28: To-Lower Result

```
hypo(true, 4, eq 'Sub1['cons['X1:Object,'X2:List]] = 'tolower[
     'Sub3['cons['X1:Object,'X2:List]]] [none] .
eq 'Sub2['cons['X1:Object,'X2:List]] = 'tolower['Sub5['cons['X1:Object,
     'X2:List]]] [none] .
eq 'Sub3['cons['X1:Object,'X2:List]] = 'X1:Object [none] .
eq 'Sub5['cons['X1:Object,'X2:List]] = 'X2:List [none] .
eq 'tolower['`[`].List] = '`[`].List [none] .
eq 'tolower['cons['X1:Object,'X2:List]] = 'cons['Sub1['cons['X1:Object,
     'X2:List]],'Sub2['cons['X1:Object,'X2:List]]] [none] .
eq 'tolower['lc['X1:Object]] = 'lc['X1:Object] [none] .
eq 'tolower['uc['X1:Object]] = 'lc['X1:Object] [none] .)
```

Listing 29: Maude Template

```
 fmod GENERATED_MODULE is

  *** SORTS
  *** basic sorts ***
  sorts InVec Int Boolean String .
  sorts Object Method .
  sorts List Item .
    subsort List < Item .
    subsort Object < Item .
    subsort Int < Object .
    subsort Boolean < Object .
    subsort String < Object .

  *** user defined sorts ***
<sorts>

  *** DT definitions (constructors)

  *** empty list
  op [] : -> List [ctor] .
  *** list concatenation
  op cons : Item List -> List [ctor] .

  *** integer "zero"
  op 0 : -> Int [ctor] .
  *** successor on Int
  op s : Int -> Int [ctor] .


  *** boolean
  op true : -> Boolean [ctor] .
  op false : -> Boolean [ctor] .


  *** defined function names (to be induced, preds)
<induce>

  *** background knowledge
<bk>

  *** input encapsulation
<invector>

  *** variables
<variables>

  *** example equations
<examples>

endfm
```

Listing 30: Code sample from the implementation of group 2

```java
/** Compares two Strings by checking whether s1 is contained in s2.
  * The comparison is case insensible by calling toLowerCase() on
  * each of the Strings.
  * @param s1 The criteria string
  * @param s2 The string which is tested whether it does contain c1
  * @return True, if s1 is contained in s2. Else return false
  */
 public static boolean compareStrings(String s1, String s2) {
   s1 = s1.trim();
   s2 = s2.trim();

   s1 = s1.toLowerCase();
   s2 = s2.toLowerCase();

   if(s2.contains(s1)) {
     return true;
   } else {
     return false;
   }
 }

 /** Checks whether the given searchTerm is contained in any String of the attributes
     array.
  * Case insensitive by applying toLowerCase. As searchTerm can still contain empty
      spaces,
  * it is split into an array of parts without empty space thus creating an array. Every
      part
  * of the search term has to be contained in one element of the attributes array. */
  public static boolean compareStringArrays(String searchTerm, String[] attributes) {
   searchTerm = searchTerm.trim();
   searchTerm = searchTerm.toLowerCase();
   String[] searchTerms = searchTerm.split(" ");

   for(String term : searchTerms) {
     if(!isTermIncludedInArray(term, attributes)) {
       return false;
     }
   }
   return true;
 }

  /** Checks whether a search term (without empty spaces) is contained in one element of
   * the attributes array. */
 private static boolean isTermIncludedInArray(String term, String[] attributes) {
   for(String att : attributes) {
     att = att.toLowerCase();
     if(att.contains(term)){
       return true;
     }
   }
   return false;
 }
```

8.2 Grammars

Listing 31: Parenthesised Expression

```

INT : '1'..'9' '0'..'9'*;
LIT : ('a'..'z')+;


value : i = INT+
        | s = LIT+
        ;

identifier
        : LIT value*
        ;

expression
  : id = identifier
  | mc = methodCall
  | li = list
  | v = value
  ;


list
  : '[' expression  (',' expression )*']'
  | '[' ']'
  ;


methodCall
  : identifier '(' argument ')'
  ;

argument
  : expression (',' expression  )*
  ;
```

Listing 32: Igor Annotation

```


INT : '0' | '1'..'9' '0'..'9'*;
LIT    : ( ('A'..'Z') | ('a'..'z') )+ ;


annotation
   : '@IgorMETA('meta');' '@IgorEQ('equations');' '@Method(' LIT+ (INT|LIT)*
        ');'
;

meta
   : method ret params
;

equations
   : 'equations={' equation* '}'
;

equation
   : '"(' exp (',' exp)* ')' '=' exp '".'
;

exp    : atom (',' atom)*
   | '[' (exp ( (',')+ exp)*)* ']'
   ;

atom   : LIT | INT ;


method
   : 'methodName="' LIT+ (INT|LIT)* '".'
;


ret
   :   'retValue="' LIT+ '".'
;


params
   : 'params="' LIT (',' LIT)* '".'
;
```