Thema: Applying The Inductive Programming System IGOR
To Flashfill-Style Enduser Programming

Bachelorarbeit

im Studiengang Angewandte Informatik der Fakultät Wirtschaftsinformatik
und Angewandte Informatik der Otto-Friedrich Universität Bamberg

Verfasser: Fabian Höpfel
Gutachterin: Prof. Dr. Ute Schmid
Abstract
In this work the inductive system IGOR will be described and how the system can be compared to the Microsoft Excel plugin Flash Fill. To show how IGOR works terms like end-user programming and inductive programming are explained at the begin of the work.

In the work the functions of the system IGOR are described and how the modules for IGOR shall be written. The examples for the IGOR modules are taken from the Microsoft Excel plugin Flash Fill. Therefore a short explanation of the Flash Fill plugin is given. The two systems will be compared in a table to show their functionalities.
## Contents

1 Introduction 1

2 Inductive Programming and Enduser Programming 2
   2.1 Inductive Programming ......................................... 2
   2.2 The IGOR System .................................................. 2
   2.3 End-user Programming ............................................. 4
   2.4 Flash Fill ......................................................... 5

3 Applying IGOR to Enduser Programming 7
   3.1 Differences between IGOR and Flash Fill ...................... 7
   3.2 Datatypes for IGOR ............................................... 8
   3.3 The Modules .................................................... 9
      3.3.1 CHANGE .................................................. 10
      3.3.2 CHANGEDATE ............................................... 12
      3.3.3 FLETTER .................................................. 14
      3.3.4 LATELTER ................................................. 16
      3.3.5 UPPER .................................................... 17

4 Comparing IGOR with Flash Fill 20

5 Conclusion and Further Work 21

References 22

A Appendix II
1 Introduction

How can people without programming knowledge adapt or program software to their needs?

In this paper this question is examined with relation to string manipulations in spreadsheets and how they can be automated. This problem is part of the *end-user programming* and to solve it an approach of the *inductive programming* is used. These concepts will be explained closer in this paper.

A famous approach of end-user programming for spreadsheets is Flash Fill, a plugin for Microsoft’s Excel. The problem of Flash Fill is, it is not open source and so we can not understand how it transforms the strings and finds the regularities. To solve this problem the system IGOR is used. IGOR is a system that uses term rewriting and pattern matching to generate recursive functional programs. This means IGOR uses input-output examples to find regularities and to generate a program from these regularities. The system IGOR allows a look inside the programs and helps to understand how these programs operate. So the research question of this paper is:

How can string transformations can be realised with the IGOR system and in what way are they comparable to Flash Fill?

A challenge is to comprehend how the input-output examples must be given to IGOR so that the system can generate a working program.

An approach to solve string transformations with IGOR is from Martin Hofman in his thesis “Automated Construction of XSL-Templates – An Inductive Programming Approach” ([Hofmann et al.]). In the approach IGOR is used as a term rewriting framework to generate XSLT stylesheets. The generated stylesheets execute functions on strings in a XML document. A prototypical program convert the input-output strings into substrings that can be read by IGOR. The substrings are recombined to generate new strings. The user pick suitable strings and completes them. Than the user compiles the specifications with the prototypical program. IGOR generates a functional program based on these specifications. A parser builds a XSLT stylesheet from the functional program.
2 Inductive Programming and Enduser Programming

2.1 Inductive Programming

“Inductive programming is the inference of an algorithm or program featuring recursive calls or repetition control structures” (Flener and Schmid, 2010a). Induction means reasoning from the specific to the general, this means you give some specific variables and try to get a solution for the general from these specific variables.

Inductive programming generates from an incomplete statement, called the evidence, and input-output examples a program. It is using recursive calls or repetitions for this purpose. If any background knowledge is needed, like predefined data-types, it is provided as an input. Examples for inductive programming are programming-by-example and programming-by-demonstration.

A workflow example of inductive programming is the transformation from input training data to the corresponding output data. The used inductive system tries to generate a program from these training data. Unlike concept learning or classification, inductive programming is restricted to algorithms and programs using recursive calls or repetition control structures.

2.2 The IGOR System

The system IGOR in the version Igor2.2 adapted to Maude is used in this paper. Maude is developed from the institut SRI International. It “is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications” (MaudeTeam).

The system allows term rewriting and the use of equational logic. Term rewriting system uses rules for term rewriting that can be compared to functions like \( f(a, b) \rightarrow c \). The equational logic allows only real equations with respect to form \( a = b \). To get the Maude system and get more information about it check the Maude homepage\(^1\).

The system IGOR uses pattern matching and term replacements to generate recursive-functional programs. This means the system finds regularities in input-output-examples and generates rules for the term replacement. To work with the Maude version of

\(^1\)http://maude.cs.uiuc.edu
IGOR, you have to load the file igor2.2.maude into Maude. You do this by starting the Maude environment and using the command “load igor2.2.maude”. You should start the Maude environment in the same location you have the IGOR files otherwise you have to use the full path to the files by loading them. These file provides everything you need to work with IGOR. As a result of loading the “igor2.2.maude”, you can start writing an instruction to IGOR for a program in the Maude environment. A better solution to write the IGOR instructions/programs is to write them in an editor and save it with file extension “.maude” and load them in the Maude environment with “load filename.maude”. To keep track of the brackets you should use an editor that shows opening and closing brackets.

To understand how IGOR ”modules” look like an example will be given. The wanted output of this program is the first item of a list.

```maude
game FIRST is
  sorts MyList MyItem InVec .
  op nil : -> MyList [ctor] .
  op first : MyList -> MyItem [metadata "induce"] .
  op in : MyList -> InVec [ctor] .
  vars A B C : MyItem .
  eq first (cons(A, nil)) = A .
  eq first (cons(A, cons(B, nil))) = A .
  eq first (cons(A, cons(B, cons(C, nil)))) = A .
endfm
```

**Code 1: Example for IGOR**

The `fmod ... is` and `endfm` box an IGOR or a Maude module/function. At the end of every command line you have to write a point.

The rest of the IGOR syntax will be explained in chapter three.

To start the IGOR synthesis you have to write

```maude
reduce in IGOR : gen( 'MODULENAME, 'FUNCTIONNAME , 'BKQ) .
```

MODULENAME is the name of the module, FUNCTIONNAME is the function/operator name of the function/operator that shall be induced. BKQ defines the background knowledge. If no background knowledge is needed
you write “noName”. If you have several background functions you write “func1 * func2 * funcn”. In this case it would look like this.

\[
\text{reduce in IGOR : gen('FNAME, 'fname, noName)} .
\]

If the input was correct IGOR will produce a HypoList with hypotheses, if more than one hypothesis is generated the hypotheses are separate from each other with “nextHypo”. This is the IGOR output for the example above.

\[
\text{reduce in IGOR : gen('FIRST, 'first, noName)} .
\text{rewrites: 483 in 0ms cpu (1ms real)} (~ \text{rewrites /second})
\]

\[
\text{result Hypo: hypo(true, 1, eq 'first ['cons['X0: MyItem, 'X1:MyList]] = 'X0:MyItem [none] .})
\]

Code 2: Output of IGOR

These hypotheses are the “programs” IGOR generated from the input-output-examples. The generated program of the example is:

\[
'\text{first ['cons['X0:MyItem, 'X1:MyList]] = 'X0:MyItem [none]}
\]

Code 3: The FIRST program

This program takes the first item X0 from a list X1 and gives the item X0 as output. So IGOR provides the “right” program. This simple example generates an easy to read output from IGOR but more complex functions generate more complex output and IGOR can generate more hypotheses. This makes it harder to understand and to read what program IGOR generated and you need some training to read the results.

### 2.3 End-user Programming

Most of the people using computers do not have programming skills. We call these people end-users and to help such end-users to adapt or develop applications the so called end-user programming or end-user development can be used. A definition of the end-user development is: “End-User Development can be defined as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artifact” (Lieberman et al., 2006).

This citation summarize the term end-user programming pretty well. End-user programming is frequently used in spreadsheets. Because spreadsheet manipulation is often used in business, the use of easy programmable programs is desirable. The work can be accomplish more efficient and faster
with the help of such programs.
A famous example of an end-user programming software for spreadsheets is Flash Fill.

2.4 Flash Fill

Flash Fill is an plugin for Microsoft Excel. It helps the user to automatically extract and continue lists. To do this you have to give an example to the left or the right side of the list, you want to manipulate, and activate Flash Fill. Normally Flash Fill starts automatically after pressing the Enter button, but you can also activate it manually in the Data tab by pressing Flash Fill.

The following examples shall explain the functionality of Flash Fill. The list contains the basketball team of the Brose Baskets Bamberg.

An easy example for Flash Fill is to extract names. The shown example extracts the first name of the players. One example suffices to get the pattern. By pressing Enter Flash Fill completes the list.

![Figure 1: First name Extraction](image)

Another example is the interchanging of the names in one column. In the Interchange column the first name and and the last name are exchanged.
Flash Fill also works with numbers. To provide an example, the date format of the birth is transformed into the American style date format.

You can see that Flash Fill finds patterns really quickly and accurate.
3 Applying IGOR to Enduser Programming

3.1 Differences between IGOR and Flash Fill

To use the examples of Flash Fill in IGOR some adaptations are needed. Flash Fill uses columns but IGOR does not understand columns because of that the columns need to be “transformed” into a format IGOR understands in this case lists. The list contains the names as a tupel consisting of first name and last name.

IGOR can not work with the predefined data types of Maude, accordingly I defined own data types for IGOR, which will be explained later.

An other problem is that IGOR does not understand strings as Flash Fill does. This is important if you need to get a single character of a string. To solve this problem I tried to teach IGOR what a string is with a new data type. The problem with this solution is that it seems IGOR in the Maude version does not understand strings longer than one character. To understand how strings are handled in Maude I read the “Maude Manual” (MaudeTeam) and checked the file “prelude.maude” which contains all predefined data types, but both sources are very vague in the definition of strings. After a lot of unsuccessful attempts, which includes the use of the Maude “special” annotation and the try to set all characters as operators, I decided to discard this approach and choose an other strategy. Instead of handling strings as a whole, this strategy handles string as a sequence of characters. This approach is not the most elegant way but it works and IGOR can work with characters of a string.

The following figure shall show how the transformation of Excel data or any spreadsheet into IGOR data shall work. The data from the spreadsheets shall be “translated” into the IGOR specification described above, after the translation IGOR generates the program and provides the solution. The solution needs to be “back translated” into normal spreadsheet representation. The whole translation procedure is not part of this work.
3.2 Datatypes for IGOR

In this section I describe the “new” defined data types for IGOR and the needed syntax for it. To define a data type in Maude/IGOR you have to declare it as a data type by writing “sort” followed by its name. To define the behavior of the data types you use the operators. To define an operator you write “op” and than how the data types shall behave. In the following part I show the four main data types of this work. At first the data type “MyString” what stands for normal strings.

\[\text{sort MyString} \ .\]

\[\text{op "} : \rightarrow \text{MyString [c t o r]} \ .\]

Code 4: Operator of MyString

The operator " defines an empty string consequent you have to write " for an empty string. This will be shown in the next section.

The next data type is the MyChar data type, which replaces the character data type. This data type is special because it is defined as a subsort of MyString. As a subsort of string it can work with the operators containing MyString. The command in Maude/IGOR is

\[\text{sorts MyString MyChar} \ .\]
subsort MyChar < MyString .

It’s operators are

\[ \text{op # : } \rightarrow \text{MyChar [ctor]} . \]
\[ \text{op ins : MyChar MyString } \rightarrow \text{MyString [ctor]} . \]

Code 5: Operators of MyChar

The operator # defines an empty string. The operator ins defines a MyChar at the start of a MyString.

The MyTupel data type stands for a tupel consisting of two strings.

\[ \text{sorts MyString MyChar MyTupel} . \]

\[ \text{op tup : MyString MyString } \rightarrow \text{MyTupel [ctor]} . \]

The operator tup defines a MyTupel to consist of two MyStrings.

The data type MyList replaces the data type list. The data type MyList has the following operators.

\[ \text{sorts MyString MyChar MyTupel MyList} . \]

\[ \text{op nil : } \rightarrow \text{MyList [ctor]} . \]
\[ \text{op cons : MyTupel MyList } \rightarrow \text{MyList [ctor]} . \]

The operator nil stands for an empty MyList. The operator cons has a MyTupel and a MyList. It joins them into a MyList, meaning it adds the MyTupel to a MyList. Why this procedure is necessary is explained in the previous section.

The expression [ctor] at the end of every operator in this section stands for constructor and means these operators are fixed. In the next section we will see that not all operators are fix. These operators need to be explained with equations. Equations are defined in Maude/IGOR with the command eq.

The data type InVec is needed for IGOR. It provides the function that IGOR can generate a program from the equations.

### 3.3 The Modules

In this section I will explain my modules for IGOR. All modules are in their full version in the appendix. The modules FNAME, LNAME, FIRSTLETTER are from a former work “Endnutzer-Programmierung zum Lernen von String-Transformationen in Excel” [Höpfel et al., 2014] and are mentioned in
the appendix only for completeness.

3.3.1 CHANGE

The module CHANGE shall provide the function to exchange first name and last name in a tuple. This module uses the MyString, MyTupel, MyList and InVec data types.

It has the mentioned operators of these data types and two extra operators. The operator change and the operator in. The change operator looks like this

\[
\text{op change} \colon \text{MyList} \rightarrow \text{MyList} \quad \text{[metadata "induce"]}.
\]

It has a MyList and goes to a MyList. The special thing about this operator is it uses not the \texttt{[ctor]} annotation but the \texttt{[metadata "induce"]} annotation. This annotation shows IGOR that this is the operator it shall provide a program for. The operator in looks like this

\[
\text{op in} \colon \text{MyList} \rightarrow \text{InVec} \quad \text{[ctor]}.
\]

It is a constructor again. It goes from MyList to InVec and is for IGOR important because without this operator it can not generate a program. The arguments before the arrow of this operator has to be equal to the arguments before the arrow of the operator with the \texttt{[metadata "induce"]} annotation or IGOR will provide an error message. Every following module has this operator.

The operator change needs equations to explain what it shall do. To produce these equations you need some variables as input. To define variables in Maude/IGOR the command is \texttt{var Name : Data Type}, if you have more variables for one data type you can also write \texttt{vars Name1 Name2 ... : Data Type}. The variables for the CHANGE module are

\[
\text{vars S T U V W X : MyString}.
\]

The equations of the CHANGE module are following.

\[
\begin{align*}
\text{eq change}(\text{cons}(\text{tup}("", \text{T}), \text{nil})) &= \text{cons}(\text{tup}(\text{T}, ""), \text{nil}) . \\
\text{eq change}(\text{cons}(\text{tup}(\text{S}, \text{T}), \text{nil})) &= \text{cons}(\text{tup}(\text{T}, \text{S}), \text{nil}) . \\
\text{eq change}(\text{cons}(\text{tup}(\text{S}, \text{T}), \text{cons}(\text{tup}(\text{U}, \text{V}), \text{nil}))) &= \text{cons}(\text{tup}(\text{T}, \text{S}), \text{cons}(\text{tup}(\text{V}, \text{U}), \text{nil})).
\end{align*}
\]
eq change(cons(tup(S, T), cons(tup(U, V), cons(tup(W, X), nil)))) = cons(tup(T, S), cons(tup(V, U), cons(tup(X, W), nil))) .

Code 6: Equations of CHANGE

The equations in the first line uses the change operator. Inside the change operator the cons operator is used to fullfill the requirement of the change operator which expect a MyList to convert it to a MyList. Inside the cons operator a tup operator is used and the operator nil to fullfill the require-ment of the cons operator which expect a MyTupel and a MyList. The tup contains the "" operator as its first MyString and the variable T as its second MyString. The result is a MyList compiled by a cons operator which contains a tup operator and nil the operator. Inside the tup the variables changed their places so that the variable T is the first MyString and the "" is the second MyString.

The second equations is similar to the first. The differneces is it uses the variable S instead of the "" operator. The third and forth equations extend the previous equations with more cons operators what means instead of one MyTupel there are more MyTupels that behaves like the one MyTupel in the first two equations. Meaning inside the MyTupel the MyStrings changes their places.

The IGOR output for this program is

reduce in IGOR : gen('CHANGE, 'change, noName)

rewrites: 18498 in 16ms cpu (17ms real)
(1156125 rewrites/second)
result Hypo: hypo(true, 2, eq 'Sub2['cons['tup ['X0:MyString, 'X1:MyString'],
'cons['tup['X2:MyString, 'X3:MyString'], 'X4: MyList ]]] = 'change['cons['tup['
'X2:MyString, 'X3:MyString'], 'X4:MyList ]] [none]

eq 'change['cons['tup['X0:MyString, 'X1:MyString ], 'nil.MyList ]] = 'cons['tup['
'X1:MyString, 'X0:MyString ], 'nil.MyList ] [none]

eq 'change['cons['tup['X0:MyString, 'X1:MyString ], 'cons['tup['X2:MyString ,
'X3:MyString], 'X4:MyList ]]] = 'cons['tup['X1: MyString, 'X0:MyString ], 'Sub2['
This output has only one hyptheses which is starting in line 3 with “result Hypo: ...”. IGOR generated a method in the CHANGE module and named it “Sub2”. This function looks like this.

\[
\text{Sub2} \left[ \text{cons} \left[ \text{tup} \left[ \text{X0} : \text{MyString} , \text{X1} : \text{MyString} \right] , \text{cons} \left[ \text{tup} \left[ \text{X2} : \text{MyString} , \text{X3} : \text{MyString} \right] , \text{X4} : \text{MyList} \right] \right] \right] = \text{change} \left[ \text{cons} \left[ \text{tup} \left[ \text{X2} : \text{MyString} , \text{X3} : \text{MyString} \right] , \text{X4} : \text{MyList} \right] \right] \ [\text{none}] .
\]

The subfunction calls the change operator on the second MyTupel in a MyList with two or more MyTupels, a recursive call. At the first equation you can see that IGOR generated the change operator as given in the equations in its module. It changes the MyString X0 with the MyString X1. The next equation shows that the program IGOR changes the MyStrings in the first MyTupel and then calls the Sub2 method, which calls the change operator for the next MyTupel. If there would be more MyTupels it would call the Sub2 method again until all MyTupels are processed.

I will give the IGOR output of the other modules for completeness and to show the programs do what they should do, but wont explain them like this IGOR output because the prodecure to read the outputs is almost the same for the different modules.

### 3.3.2 CHANGEDATE

The next module is the CHANGEDATE module. This module provides the function to switch the date format, e.g. the american date format into the european or vice versa.

The module has the data types MyString, MyList and InVec. It also has a new data type called MyDate. This data type shall represent the normal data format containing of three numbers or three strings. To achieve this three new operators are created. At first the date operator

\[
\text{op date} : \text{MyString MyString MyString} \rightarrow \text{MyDate} [\text{ctor}] .
\]

It simple defines that the three MyStrings in the operator are the MyDate. Than the conD operator
It generates from a MyDate MyList a MyList, quite the same what the \textit{cons} operator of \textit{do} for MyTupel MyList. This operator is needed to change the date format of more MyDates in a MyList.

And the main operator \textit{changeD}:

\begin{center}
\begin{align*}
\text{op changeD} & : \text{MyList} \rightarrow \text{MyList} [\text{metadata "induce"}] .
\end{align*}
\end{center}

The main operator that moves a MyList to a MyList.

The variables for the \textit{CHANGEDATE} module are

\begin{center}
\begin{align*}
\text{vars A B C D E F G H I} & : \text{MyString} .
\end{align*}
\end{center}

As you can see all are MyString variables. That is because you can write numbers also as strings and both the numeric and the string representation can be achieved this way.

The equations for \textit{CHANGEDATE}:

\begin{center}
\begin{align*}
\text{eq changeD}(\text{conD}(\text{date("", ",", ",")}, \text{nil})) &= \text{conD}(
\text{date("", ",", ",")}, \text{nil}) .
\text{eq changeD}(\text{conD}(\text{date}(\text{A, B, C}), \text{nil})) &= \text{conD}(
\text{date}(\text{B, A, C}), \text{nil}) .
\text{eq changeD}(\text{conD}(\text{date}(\text{A, B, C}), \text{conD}(\text{date}(\text{D, E, F}), \text{nil}))) &= \text{conD}(\text{date}(\text{B, A, C}), \text{conD}(\text{date}(\text{E, D, F}), \text{nil})) .
\text{eq changeD}(\text{conD}(\text{date}(\text{A, B, C}), \text{conD}(\text{date}(\text{D, E, F}), \text{conD}(\text{date}(\text{G, H, I}), \text{nil})))) &= \text{conD}(\text{date}(\text{B, A, C}), \text{conD}(\text{date}(\text{E, D, F}), \text{conD}(\text{date}(\text{H, G, I}), \text{nil}))) .
\end{align*}
\end{center}

The first equation simply uses empty strings, so I will skip this equation and go directly to the second equation. This equation uses the \textit{changeD} operator. This operator expects a MyList which is generated by the \textit{conD} operator. The expected MyDate is generated by the \textit{date} operator and the MyList by the \textit{nil} operator. The \textit{date} operator uses the variables A, B and C in this order. The result is a MyList generated by a \textit{conD}. The \textit{date} looks in the result this way B, A, C. Meaning the first and the second MyString changes their places. The remaining equations have the same pattern only with more MyDates in a MyList.

The IGOR output

\begin{center}
\begin{align*}
\text{reduce in IGOR} & : \text{gen( 'CHANGEDATE, 'changeD, noName) .}
\end{align*}
\end{center}
rewrites: 72118 in 88ms cpu (88ms real) (819522 rewrites/second)
result Hypo: hypo(true, 2, eq 'Sub2[ 'conD[ 'date['X0:MyString, 'X1:MyString, 'X2:MyString], 'conD[ 'date['X3:MyString, 'X4:MyString, 'X5:MyString], 'X6:MyList]] = 'changeD[ 'conD[ 'date['X3:MyString, 'X4:MyString, 'X5:MyString, 'X6:MyList]]]
    'conD[ 'date['X1:MyString, 'X0:MyString, 'X2:MyString], 'nil. MyList][none] .
    eq 'changeD[ 'conD[ 'date['X0:MyString, 'X1:MyString, 'X2:MyString], 'nil. MyList][none] .
    eq 'changeD[ 'conD[ 'date['X0:MyString, 'X1:MyString, 'X2:MyString], 'conD[ 'date['X3:MyString, 'X4:MyString, 'X5:MyString], 'X6:MyList]] = 'conD[ 'date['X1:MyString, 'X0:MyString, 'X2:MyString], 'nil. MyList][none] .

Code 8: IGOR output for CHANGEDATE

3.3.3 FLETTER

The FLETTER module shall get from two MyStrings in a MyTupel the first letter of the first MyString and the full second MyString. This shall be the process of getting the initial of the first name and the full last name.

The used data types are MyString, MyChar, MyList, MyTupel and InVec. The special operators for this module are the firstletter and the fletter.

\[
\begin{align*}
\text{op firstletter : MyString} & \rightarrow \text{MyChar} \text{ ctor } . \\
\text{op fletter : MyList} & \rightarrow \text{MyList} \text{ metadata } "\text{ induce }\). 
\end{align*}
\]

The firstletter operator is from the previous work “Endnutzer-Programmierung zum Lernen von String-Transformationen in Excel” (Höpfel et al., 2014) and produce from a MyString a MyChar. The “main” operator fletter produce from a MyList a MyList again.

The used variables are
vars S T U V W X : MyString .
vars E F G : MyChar .

In this module we have two kinds of variables. The MyString variables and the MyChar variables. The two kinds of variables are necessary because the module shall generate a program that gives a MyChar followed by a MyString in a MyTupel.

The equations for this module are

\[
\text{eq } f\text{lette} \text{r}(\text{cons}(\text{tup}(\text{firstletter}(\text{ins}(E, S)), T), \text{nil})) = \text{cons}(\text{tup}(E, T), \text{nil}) .
\]

\[
\text{eq } f\text{lette} \text{r}(\text{cons}(\text{tup}(\text{firstletter}(\text{ins}(E, S)), T), \text{cons}(\text{tup}(\text{firstletter}(\text{ins}(F, U)), V), \text{nil}))) = \text{cons}(\text{tup}(E, T), \text{cons}(\text{tup}(F, V), \text{nil})) .
\]

\[
\text{eq } f\text{lette} \text{r}(\text{cons}(\text{tup}(\text{firstletter}(\text{ins}(E, S)), T), \text{cons}(\text{tup}(\text{firstletter}(\text{ins}(F, U)), V), \text{cons}(\text{tup}(\text{firstletter}(\text{ins}(G, W)), X), \text{nil})))) = \text{cons}(\text{tup}(E, T), \text{cons}(\text{tup}(F, V), \text{cons}(\text{tup}(G, X), \text{nil}))) .
\]

The first equation generates the MyList for the \texttt{fletter} operator from the \texttt{cons} operator that has the operators \texttt{tup} and \texttt{nil} inside. The \texttt{tup} operator consists of the \texttt{firstletter} operator and the variable \texttt{T}. Inside of the \texttt{firstletter} is the \texttt{ins} operator. The \texttt{ins} operator generates a MyString from a MyChar \texttt{E} and a MyString \texttt{S}. The result is a MyList generated from a \texttt{cons} operator. The MyList consists of a MyTupel and the \texttt{nil} operator. The MyTupel has a MyChar \texttt{E} and the variable \texttt{T}. The \texttt{tup} operator expect a MyString but gets a MyChar with the \texttt{E} variable. This is possible because MyChar is a subsort of MyString and inherits the methods from MyString.

The output of IGOR is very long in this case, because IGOR generates multiple hypotheses. I will show only the first hypothesis and the full output in the appendix.
3.3.4 LALETTER

The LALETTER module shall get from two MyStrings in a MyTupel the full first MyString and the first letter of the second MyString. This shall be the process of getting the full first name and the intial of the last name. I will not explain this module LALETTER because it is quiet the same as the FLETTER module as you can see.

To show that the IGOR system can produce a program for this program too, I will show the output. The produced output of IGOR is again very long. I will only show the first hypothesis again and put the full output into the appendix.

reduce in IGOR: gen('LALETTER, 'laletter', noName).
rewrites: 681491 in 668ms cpu (667ms real)
(1020196 rewrites/second)
result HypoList: hypo(true, 3, eq 'laletter['
  cons['tup['X0:MyString,'firstletter['""'.MyString]],'nil.MyList']] = '

Code 9: IGOR output for FLETTER


Code 10: IGOR output for LALETTER

3.3.5 UPPER

The last module is the UPPER module. It shall get the upper cases of words to produce shortcuts, e.g. for “Association for Computing Machinery” it shall produce “ACM”. In other words it shall get the new data type MyUpper from a MyString.

The UPPER module uses the data types MyString, MyChar, MyLower, MyUpper, MyList and InVec. The data type MyUpper shall represent upper case letters and the data type MyLower the lower case letters. Both data types are subsorts of MyChar.

The MyUpper data type has two special operators.

\[
\text{op foo : } \rightarrow \text{MyUpper [ setor ] .}
\]

\[
\text{op result : MyUpper MyList } \rightarrow \text{MyList [ setor ] .}
\]

The foo operator is used for an uniform IGOR output, you could use the # operator aswell but IGOR would write in the equation of the output X0.MyChar instead of X0.MyUpper. The result operator generates a MyList from a Myupper and a MyList.

Another new operator is the conC operator, it generates a MyList from a MyString and a MyList.
The main operator \( upper \) produces like the other main operators a MyList from a MyList. The used variables are

\[
\text{var } S : \text{MyString} \ . \ \text{vars } A, B : \text{MyUpper} \ . \ \text{var } L : \text{MyLower} \ .
\]

The equations of the \( UPPER \) module

\[
\begin{align*}
\text{eq } & \text{upper} (\text{conC} (\text{ins} (\text{foo}, \text{""}), \text{nil})) = \text{result} (\text{foo}, \text{nil}) \\
\text{eq } & \text{upper} (\text{conC} (\text{ins} (A, \text{""}), \text{nil})) = \text{result} (A, \text{nil}) \\
\text{eq } & \text{upper} (\text{conC} (\text{ins} (A, S), \text{nil})) = \text{result} (A, \text{nil}) \\
\text{eq } & \text{upper} (\text{conC} (\text{ins} (A, \text{ins} (L, S)), \text{nil})) = \text{result} (A, \text{nil}) \\
\text{eq } & \text{upper} (\text{conC} (\text{ins} (A, \text{ins} (B, S)), \text{nil})) = \text{result} (A, \text{result} (B, \text{nil})) \\
\text{eq } & \text{upper} (\text{conC} (\text{ins} (L, \text{ins} (A, S)), \text{nil})) = \text{result} (A, \text{nil})
\end{align*}
\]

The first three equations behave almost same only the variables respectively the operators for empty string/empty MyUpper are different. So I will explain the third equations what should explain the other two aswell. The \( upper \) operator expects a MyList which is produced by the \( \text{conC} \) operator. The \( \text{conC} \) operator has a MyString and the empty MyList. The MyString is generated by the \( \text{ins} \) operator which has the MyUpper \( A \) and the MyString \( S \) inside. The result is a MyList produced by the \( \text{result} \) operator which has the MyUpper \( A \) and the empty MyList. So it extracted the MyUpper \( A \) from the MyString \( \text{ins}(A, S) \).

The last two equations have caused problems with IGOR. If you comment them out everything works fine. If they are not comment out, IGOR will not generate a program. I could not figure out why this happens, because the syntax is right. It seems that IGOR can not access the second \( \text{ins} \) operator.

The IGOR output without the two “problem” equations for the \( UPPER \) module is

\[
\begin{align*}
\text{reduce in IGOR : gen('UPPER, 'upper, noName) .} \\
\text{rewrites: 1447 in 4ms cpu (4ms real) (361750 rewrite/second)} \\
\text{result Hypo: hypo(true, 1, eq 'upper ['conC['ins ['X0:MyUpper, 'X1:MyString],}
\end{align*}
\]
\texttt{\texttt{'nil.MyList}}} = \texttt{\texttt{\texttt{result ['X0:MyUpper, 'nil.MyList}}} \\
\texttt{\texttt{\texttt{\texttt{\texttt{none}}} .}})

Code 11: IGOR output for UPPER
4 Comparing IGOR with Flash Fill

This section will show the comparison of IGOR and Flash Fill. The different functions will be listed in a table.

For better understanding the functions are described shortly. The first three functions are as mentioned from the work “Endnutzer-Programmierung zum Lernen von String-Transformationen in Excel” (Höpfel et al., 2014).

The “Get First Letter” function extracts the first letter of a string. The “Get First Name” extracts the first name from the full name and the “Get Last Name” the last name of the full name.

The “Swap Names” function exchanges the first name for the last name. The “Change Date Format” changes the date format from american date format into the european or vice versa. The “Get Initial Of First Name And Full Last Name” function extract from first name and last name the initial of the first name and the full last name. The “Get Full First Name And Initial Of Last Name” does the same for the last name. The “Get Upper Case Letters” gets the upper letters in a string, e.g. for shortcuts.

As you can see in the table, IGOR was able to solve all the given examples.

<table>
<thead>
<tr>
<th>Functions</th>
<th>IGOR</th>
<th>Flash Fill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get First Letter</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Get First Name</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Get Last Name</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Swap Names</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Change Date Format</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Get Initial Of First Name And Full Last Name</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Get Full First Name And Initial Of Last Name</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Get Upper Case Letters</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
5 Conclusion and Further Work

The goal of this work was to show that IGOR can produce the same programs for end-users that Flash Fill provides them. To achieve this goal in the begin of this work the base knowledge, like what is end-user programming or what is IGOR and how does it work, was explained.

The essence of the work was to find input-output examples for IGOR and produce the IGOR modules for them. As this work shows IGOR can generate solutions to the selected examples the same way that Flash Fill can. And I am sure IGOR can find solution for all Flash Fill functions with the right input-output examples. IGOR should be more powerfull than Flash Fill so it should even find solutions for problems Flash Fill can not solve. A problem from the Maude version of IGOR is the really scarce and unspecific error output. For example if you have an error in a module it can show you the line number inside the module, the line number of the whole code or it simply doesnt show you where the error is. Sometimes it can happen that loading a module to Maude does not cause any error but trying to generate a program with IGOR cause an error output because of the syntax.

In further work the realisation of transforming the input from Excel tables to IGOR specifications shall be realised and if possible automated. In the “backtranslating process” the user should be able to choose a program if IGOR produces more than one hypothesis.

To help the “developers” it would be very helpful to generate a more specific error output or a visual developer environment for Maude/IGOR.

Another point that should be cleared is the problem with the strings. If IGOR could understand strings the \textit{UPPER} module would work with real strings and not with a chain of MyChars.

In conclusion the IGOR system is a mighty tool for end-user programming and it is quite interesting to work with.
References


List of Figures

1 First name Extraction ........................................ 5
2 Interchange names ............................................. 6
3 Date format change ........................................... 6
4 Excel to IGOR .................................................. 8
List of Tables

1 Comparison of IGOR and Flashfill . . . . . . . . . . . . . . . . . . . . . 20
List of Code

1  Example for IGOR ........................................ 3
2  Output of IGOR ........................................... 4
3  The FIRST program ....................................... 4
4  Operator of MyString ..................................... 8
5  Operators of MyChar ...................................... 9
6  Equations of CHANGE .................................... 10
7  IGOR output for CHANGE ................................. 11
8  IGOR output for CHANGEDATE ......................... 13
9  IGOR output for FLETTER .................. ............. 15
10 IGOR output for LALETTER ......................... 16
11 IGOR output for UPPER ............................... 18
ABSTRACT
A Appendix

CD content

The attached CD contains the file *spreadsheetproblems.maude* with the IGOR modules, the used IGOR version and a copy of this work.

The IGOR modules

```maude
fmod FNAME is
  sorts MyString MyTupel MyList InVec .

  op "" : → MyString [ctor] .
  op nil : → MyList [ctor] .
  op result : MyString MyList → MyList [ctor] .
  op in : MyList → InVec [ctor] .
  op fname : MyList → MyList [metadata "induce"] .

  vars S T U V W X Y Z A B C D : MyString .

  eq fname(cons(tup(S,T),nil)) = result(S,nil) .
  eq fname(cons(tup(S,T),cons(tup(U,V),nil))) = result(S,result(U,nil)) .
  eq fname(cons(tup(S,T),cons(tup(U,V),cons(tup(W,X),nil)))) = result(S,result(U,result(W,nil))) .
  eq fname(cons(tup(S,T),cons(tup(U,V),cons(tup(W,X),cons(tup(Y,Z),nil))))) = result(S,result(U,result(W,result(Y,nil)))) .
  eq fname(cons(tup(S,T),cons(tup(U,V),cons(tup(W,X),cons(tup(Y,Z),cons(tup(A,B),nil)))))) = result(S,result(U,result(W,result(Y,result(A,nil))))) .
  eq fname(cons(tup(S,T),cons(tup(U,V),cons(tup(W,X),cons(tup(Y,Z),cons(tup(A,B),cons(tup(C,D),nil))))))) = result(S,result(U,result(W,result(Y,result(A,result(C,nil)))))) .
```

II
endfm

fmod LNAME is
  sorts MyString MyTupel MyList InVec .

  op nil : -> MyList [ctor] .
  op result : MyString MyList -> MyList [ctor] .
  op in : MyList -> InVec [ctor] .
  op lname : MyList -> MyList [metadata "induce"] .

  vars S T U V W X Y Z A B C D : MyString .

  eq lname (cons (tup (S, T), nil)) = result (T, nil) .
  eq lname (cons (tup (S, T), cons (tup (U, V), nil))) = result (T, result (V, nil)) .
  eq lname (cons (tup (S, T), cons (tup (U, V), cons (tup (W, X), nil)))) = result (T, result (V, result (X, nil))) .
  eq lname (cons (tup (S, T), cons (tup (U, V), cons (tup (W, X), cons (tup (Y, Z), nil))))) = result (T, result (V, result (X, result (Z, nil)))) .
  eq lname (cons (tup (S, T), cons (tup (U, V), cons (tup (W, X), cons (tup (Y, Z), cons (tup (A, B), nil))))) = result (T, result (V, result (X, result (Z, result (B, nil))))) .
  eq lname (cons (tup (S, T), cons (tup (U, V), cons (tup (W, X), cons (tup (Y, Z), cons (tup (A, B), cons (tup (C, D), nil))))) = result (T, result (V, result (X, result (Z, result (B, result (D, nil))))))) .

endfm

fmod FIRSTLETTER is
  sorts MyString MyChar InVec .

  ***op tostring : MyChar -> MyString [ctor] .

III
op # : $\rightarrow$ MyChar [ctor] .

***op + : MyString MyString $\rightarrow$ MyString [ctor] .

op ins : MyChar MyString $\rightarrow$ MyString [ctor] .
op in : MyString $\rightarrow$ InVec [ctor] .
op firstletter : MyString $\rightarrow$ MyChar [metadata "induce"] .

var S : MyString . var A : MyChar .

eq firstletter("") = # .
eq firstletter(ins(A,"")) = A .
eq firstletter(ins(A,S)) = A .

endfm

fmod CHANGE is
sorts MyString MyTupel MyList InVec .

op "" : $\rightarrow$ MyString [ctor] .
op nil : $\rightarrow$ MyList [ctor] .
op tup : MyString MyString $\rightarrow$ MyTupel [ctor] .
op cons : MyTupel MyList $\rightarrow$ MyList [ctor] .
op in : MyList $\rightarrow$ InVec [ctor] .
op change : MyList $\rightarrow$ MyList [metadata "induce "] .

vars S T U V W X : MyString .

eq change(cons(tup("", T), nil)) = cons(tup(T, ""), nil) .
eq change(cons(tup(S, T), nil)) = cons(tup(T, S ), nil) .
eq change(cons(tup(S, T), cons(tup(U, V), nil)) ) = cons(tup(T, S), cons(tup(V, U), nil)) .
eq change(cons(tup(S, T), cons(tup(U, V), cons( tup(W, X), nil)))) = cons(tup(T, S), cons( tup(V, U), cons(tup(X, W), nil))) .

endfm
American to European or European to American

fmod CHANGEDATE is
sorts MyString MyDate MyList InVec.

op "": -> MyString [ctor].
op date: MyString MyString MyString -> MyDate [ctor].
op nil: -> MyList [ctor].
op conD: MyDate MyList -> MyList [ctor].
op changeD: MyList -> MyList [metadata "induce"].
op in: MyList -> InVec [ctor].

vars A B C D E F G H I: MyString.

eq changeD(conD(date("", "", ""), nil)) = conD(date("", "", ""), nil).
eq changeD(conD(date(A, B, C), nil)) = conD(date(B, A, C), nil).
eq changeD(conD(date(A, B, C), conD(date(D, E, F), nil))) = conD(date(B, A, C), conD(date(E, D, F), nil)).
eq changeD(conD(date(A, B, C), conD(date(D, E, F), conD(date(G, H, I), nil)))) = conD(date(B, A, C), conD(date(E, D, F), conD(date(H, G, I), nil))).

endfm

fmod FLETTER is
sorts MyString MyTupel MyList MyChar InVec.
subsort MyChar < MyString.

*** op toString : MyChar -> MyString [ctor].
op "": -> MyString [ctor].
op #: -> MyChar [ctor].
op ins: MyChar MyString -> MyString [ctor].
op nil: -> MyList [ctor].
op tup: MyString MyString -> MyTupel [ctor].
op cons: MyTupel MyList -> MyList [ctor].
op firstLetter: MyString -> MyChar [ctor].
\begin{verbatim}

op fletter : MyList → MyList [metadata "induce"] .

op in : MyList → InVec [ctor] .

vars S T U V W X : MyString .
vars E F G : MyChar .

eq fletter (cons (tup (firstletter (ins (E, S)), T), nil)) = cons (tup (E, T), nil) .

eq fletter (cons (tup (firstletter (ins (E, S)), T),
cons (tup (firstletter (ins (F, U)), V), nil)))
= cons (tup (E, T), cons (tup (F, V), nil)) .

eq fletter (cons (tup (firstletter (ins (E, S)), T),
cons (tup (firstletter (ins (F, U)), V),
cons (tup (firstletter (ins (G, W)), X), nil))))
= cons (tup (E, T), cons (tup (F, V), cons (tup (G, X), nil))) .

endfm

fmod LALETTER is
sorts MyString MyTupel MyList MyChar InVec .
subsort MyChar < MyString .

***op tostring : MyChar → MyString [ctor] .
op "" : → MyString [ctor] .
op # : → MyChar [ctor] .
op ins : MyChar MyString → MyString [ctor] .
op nil : → MyList [ctor] .
op firstletter : MyString → MyChar [ctor] .
op laletter : MyList → MyList [metadata "induce"] .

op in : MyList → InVec [ctor] .

vars S T U V W X : MyString .
vars E F G : MyChar .

VI
\end{verbatim}
eq laletter (cons (tup (T, firstletter (ins (E, S))) , nil)) = cons (tup (T, E), nil).

eq laletter (cons (tup (T, firstletter (ins (E, S))) , cons (tup (V, firstletter (ins (F, U))) , nil))) = cons (tup (T, E), cons (tup (V, F), nil)).

endfm

fmod UPPER is
  sorts MyString MyChar MyLower MyUpper MyList
    InVec .
  subsort MyChar < MyString .
  subsort MyLower < MyChar .
  subsort MyUpper < MyChar .

  op # : -> MyChar [ctor] .
  op nil : -> MyList [ctor] .
  op conC : MyString MyList -> MyList [ctor] .
  op upper : MyList -> MyList [metadata "induce"] .

  op in : MyList -> InVec [ctor] .

  var S : MyString . vars A B : MyUpper . var L : MyLower .

  eq upper (conC (ins (foo, ""), nil)) = result (foo, nil).
  eq upper (conC (ins (A, ""), nil)) = result (A, nil).
  eq upper (conC (ins (A, S), nil)) = result (A, nil).

VII
eq upper(conC(ins(A, ins(L, S)), nil)) = result(A, nil).

***eq upper(conC(ins(A, ins(B, S)), nil)) = result(A, result(B, nil)).

***eq upper(conC(ins(L, ins(A, S)), nil)) = result(A, nil).

endfm

IGOR outputs

FLETTER module

reduce in IGOR : gen('FLETTER, 'fletter , noName).

rewrites: 64774 in 76ms cpu (77ms real) (852289 rewrites/second)

result HypoList: hypo(true, 2, eq 'Sub2['cons['
tup['firstletter['ins['X0:MyChar,'X1:MyString'],']cons['
tup['firstletter['ins['X3:MyChar,'X4:MyString'],']cons['
'X5:MyString'],']X6:MyList']
']cons['
'tup['firstletter['ins['X3:MyChar,'X1:MyString']
']',']X5:MyString'],']X6:MyList']
']none].

eq 'fletter['cons['
tup['firstletter['ins['X0:MyChar,'X1:MyString']
',['X2:MyString'],']nil.MyList']
']cons['
tup['X0:MyChar,'X2:MyString'],']nil.MyList']
']none].

eq 'fletter['cons['
tup['firstletter['ins['X0:MyChar,'X1:MyString']
',['X2:MyString'],']cons['
tup['firstletter['ins['X3:MyChar,'X4:MyString']
',['X5:MyString'],']cons['
'tup['firstletter['ins['X3:MyChar,'X4:MyString']
',['X5:MyString'],']X6:MyList']
']cons['
'tup['firstletter['ins['X0:MyChar,'X1:MyString']
',['X2:MyString'],']cons['
'tup['firstletter['ins['X3:MyChar,'X4:MyString']
',['X5:MyString'],']X6:MyList']
']none].

VIII
nextHypo

hypo(true, 2, eq 'Sub2['cons['tup['firstletter
'ins['X0:MyChar,'X1:MyString]','X2:MyString],'cons['tup['firstletter['ins['X3:
MyChar,'X4:MyString]','X5:MyString]','X6:MyList']]'eq 'letter['cons['
tup['firstletter['ins'
'X3:MyChar,'X2:MyString]','X5:MyString]','X6:MyList']][none].

eq 'letter['cons['tup['firstletter['ins['X0:
MyChar,'X1:MyString]','X2:MyString]','nil.MyList']]'eq 'cons['tup['X0:
MyChar,'X2:MyString]','nil.MyList'][none].

eq 'letter['cons['tup['firstletter['ins['X0:
MyChar,'X1:MyString]','X2:MyString]','cons['
tup['firstletter['ins['X3:MyChar,'X4:MyString]','X5:MyString]','X6:MyList']]'Sub2[
'cons['tup['firstletter['ins['X0:MyChar,'X1:MyString]','X2:MyString]','X5:MyString]','X6:MyList']][none].

nextHypo

hypo(true, 2, eq 'Sub2['cons['tup['firstletter
'ins['X0:MyChar,'X1:MyString]','X2:MyString]','cons['tup['firstletter['ins['X3:
MyChar,'X4:MyString]','X5:MyString]','X6:MyList']]'eq 'letter['cons['
tup['firstletter['ins'
'X3:MyChar,'X4:MyString]','X5:MyString]','X6:MyList']][none].

eq 'letter['cons['tup['firstletter['ins['X0:
MyChar,'X1:MyString]','X2:MyString]','nil.MyList']]'eq 'cons['tup['X0:
MyChar,'X2:MyString]','nil.MyList'][none].

IX
'X2 : MyString]' , 'nil . MyList]' ] = 'cons[ 'tup[ 'X0: MyChar, 'X2: MyString]' ,
eq 'fletter[ 'cons[ 'tup[ 'first letter[ 'ins[ 'X0: MyChar, 'X1: MyString]' ,
'X2: MyString]' , 'cons[ 'tup[ 'first letter[ 'ins[ 'X3: MyChar, 'X4: MyString]' ,
'X5: MyString]' , 'X6: MyList]' ] ] = 'cons[ 'tup[ 'X0: MyChar, 'X2: MyString]' , 'Sub2[ 'cons[ 'tup[ 'first letter[ 'ins[ 'X0: MyChar, 'X1: MyString]' ,
'X2: MyString]' , 'cons[ 'tup[ 'first letter[ 'ins[ 'X3: MyChar, 'X4: MyString]' ,
'X5: MyString]' , 'X6: MyList]' ] ] , 'X5: MyString]' ,
'X6: MyList]' ] ] [ none ] .

nextHypo

hypo( true , 2 , eq 'Sub2[ 'cons[ 'tup[ 'first letter[ 'ins[ 'X0: MyChar, 'X1: MyString]' ,
'X2: MyString]' , 'cons[ 'tup[ 'first letter[ 'ins[ 'X3: MyChar, 'X4: MyString]' ,
eq 'fletter[ 'cons[ 'tup[ 'first letter[ 'ins[ 'X0: MyChar, 'X1: MyString]' ,
'X2: MyString]' , 'nil . MyList]' ] = 'cons[ 'tup[ 'X0: MyChar, 'X2: MyString]' ,
eq 'fletter[ 'cons[ 'tup[ 'first letter[ 'ins[ 'X0: MyChar, 'X1: MyString]' ,
'X2: MyString]' , 'cons[ 'tup[ 'first letter[ 'ins[ 'X3: MyChar, 'X4: MyString]' ,
'X5: MyString]' , 'X6: MyList]' ] ] = 'cons[ 'tup[ 'X0: MyChar, 'X2: MyString]' , 'Sub2[ 'cons[ 'tup[ 'first letter[ 'ins[ 'X0: MyChar, 'X1: MyString]' ,
'X2: MyString]' , 'cons[ 'tup[ 'first letter[ 'ins[ 'X3: MyChar, 'X4: MyString]' ,
'X5: MyString]' , 'X6: MyList]' ] ] , 'X5: MyString]' ,
'X6: MyList]' ] ] [ none ] .)
LALETTER module

reduce in IGOR : gen( 'LALETTER', 'laletter', noName ) .

rewrites: 61943 in 116ms cpu (114ms real)
(533991 rewrites/second)

nextHypo


XI
nextHypo

hypo( true , 2 , eq 'Sub2 [' 'cons [' 'tup [' 'X0: MyString ,
   'firstletter [' 'ins [' 'X1: MyChar ,
   'X2: MyString ]] ]
   , 'cons [' 'tup [' 'X3: MyString ,
   'firstletter [' 'ins [' 'X4: MyChar ,
   'X5: MyString ]] ]
   , 'X6: MyList ] ]
   = 'cons [' 'tup [' 'X0: MyString ,
   'X1: MyChar ] ,
   'Sub2 [' 'cons [' 'tup [' 'X3: MyString ,
   'firstletter [' 'ins [' 'X4: MyChar ,
   'X5: MyString ]] ]
   , 'X6: MyList ] ]
   [ none ] .
)

'firstletter [' 'ins [' 'X4: MyChar , 'X2: MyString ] ]
eq 'laletter [' 'cons [' 'tup [' 'X0: MyString ,
   'firstletter [' 'ins [' 'X1: MyChar ,
   'X2: MyString ] ]
   , 'cons [' 'tup [' 'X3: MyString ,
   'firstletter [' 'ins [' 'X4: MyChar ,
   'X5: MyString ]] ]
   , 'X6: MyList ] ]
   = 'cons [' 'tup [' 'X0: MyString ,
   'X1: MyChar ] ,
eq 'laletter [' 'cons [' 'tup [' 'X0: MyString ,
   'firstletter [' 'ins [' 'X1: MyChar ,
   'X2: MyString ] ]
eq 'laletter [' 'cons [' 'tup [' 'X0: MyString ,
   'firstletter [' 'ins [' 'X1: MyChar ,
   'X2: MyString ] ]
   , 'cons [' 'tup [' 'X3: MyString ,
   'firstletter [' 'ins [' 'X4: MyChar ,
   'X5: MyString ] ]
   , 'X6: MyList ] ]
   = 'cons [' 'tup [' 'X0: MyString ,
   'X1: MyChar ] ,
   'Sub2 [' 'cons [' 'tup [' 'X3: MyString ,
   'firstletter [' 'ins [' 'X4: MyChar ,
   'X5: MyString ]] ]
   , 'X6: MyList ] ]
   [ none ] . )

XII
Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Datum)  (Unterschrift)