Otto-Friedrich-Universität Bamberg
Fakultät WIAI

Kognitive Systeme

Ausarbeitung

im Rahmen der Veranstaltung

KogSys-Projekt 2014

zum Thema:

Endnutzer-Programmierung zum Lernen von String-Transformationen in Excel;

Ein Vergleich des Excel Plugins Flash Fill mit dem induktiven Programmiersystem Igor

Vorgelegt von:

Fabian Höpfel
Peter Hohmann
Alexandra Rohm

Betreuer: Ute Schmid

Bamberg, Sommersemester 2014

Inhaltsverzeichnis

1 Einführung

In der vorliegenden Arbeit wird untersucht, wie spezielle Probleme aus dem Bereich der *Endnutzer-Programmierung* mithilfe eines Ansatzes zur induktiven Programmsynthese gelöst werden können. Endnutzer-Programmierung ist ein Forschungsgebiet der Informatik. Es befasst sich damit, wie es dem Endnutzer ermöglicht werden kann, Software an seine Bedürfnisse anzupassen ohne über Programmierkenntnisse zu verfügen. Eines der erfolgreichsten End User Programming-Systeme ist das Excel Plug-in Flash Fill, welches im Verlauf dieser Arbeit genauer erläutert wird. Ein Teilgebiet des End User Programming ist die induktive Programmsynthese, also die automatische Entwicklung rekursiver Programme aus Input/Output-Beispielen. Dabei beobachtet das System Gesetzmäßigkeiten in der Benutzereingabe und generalisiert daraus eigenständig die Ausgabe. Das heißt, dass Programme aus Beobachtungen erlernt werden. Im speziellen Fall dieser Arbeit sollen Stringtranformationen mithilfe von „programing by example“ gelernt werden. Das dazu verwendete Tool ist Igor 2, in einer für Maude, eine funktionale Programmiersprache, angepassten Version. Auf beide Tools, Igor und Maude, wird im Folgenden weiter eingegangen.
Die Forschungsfragen dieser Arbeit lauten:
Wie können String-Transformationen mit Igor realisiert werden? Ist diese Leistung mit der von Flash Fill vergleichbar?
Da es sich bei Flash Fill um eine sogenannte „Closed Source“ handelt, ist Flash Fills Quellcode nicht verfügbar. Dies macht es interessant die zugrundeliegende Vorgehensweise zu verstehen. Daher wird versucht das allgemeine System IGOR anzuwenden um Stringtransformationen zu erlernen. Eine wesentliche Herausforderung ist, zu ermitteln welche Art von I/O-Beispielen Igor benötigt, um Transformationen zu lernen. Im Voraus müssen jedoch zuerst die Grundlagen funktionaler Programmierung klar werden. Weiterhin wird ein Überblick über algebraische Datentypen gegeben und eine Einführung in die Funktionsweise von Maude und Igor, mit Beispielen. Dieses Vorgehen hilft die Funktionsweise von Maude zu verstehen und somit leichter Input-Output-Beispiele für IGOR zu finden. Um den Umgang mit Maude und IGOR zu vereinfachen wurde eine grafische Oberfläche realisiert. Diese GUI soll bei der Analyse des Codes sowie der Ergebnisse helfen und ein leichteres Arbeiten mit Maude/IGOR ermöglichen. Schließlich werden verschiedene String-Transformationsprobleme basierend auf algebraischen Datentypen als Input-/Output-Beispiele für IGOR präsentiert und geprüft ob IGOR die gewünschten Funktionalitäten erlernen kann.

2 Endnutzeprogrammierung

2.1 Grundkonzept der Stringtransformation nach M. Hofmann

Es gab bereits Versuche Stringtransformationen durchzuführen. Einer davon ist die Arbeit von Martin Hofmann. In dieser Arbeit wurde gezeigt, dass es möglich ist aus wenigen Input/Output Beispielen automatisch XSLT Stylesheets zu generieren, indem man IGOR als term rewriting Framework nutzt. Die generierten Stylesheets führen einfache Funktionen auf Strings in XML Dokumenten aus. Ein prototypisches Programm wandelt die Input/Output Strings in eine Liste von Substrings als zugrunde liegende Datenstruktur um, damit diese als geeignete Beispiele für IGOR verwendet werden können. Durch Rekombination der Substrings der Eingabe-Strings werden weitere Strings generiert. Der Benutzer wählt daraus geeignete Strings aus und vervollständigt diese und erstellt mithilfe des Prototypen die Spezifikationen für IGOR, welches ein funktionales Programm gemäß der Spezifikationen erstellt. Zum Schluss erstellt ein Parser aus den funktionalen Programm eine XSLT Stylesheet.

2.2 Flash Fill

Flash Fill ist ein Feature von Microsoft Excel 2013. Die deutsche Übersetzung für Flash Fill ist „Blitzvorschau“.
Es basiert auf einer „domainspecific language“ und erlaubt, dass die Bearbeitung von Excel-Tabellen durch einfache Programme automatisiert wird. Diese werden aus nur wenigen Benutzereingaben inferiert. Es führt im wesentlichen String-Transformationen durch. Weiterhin ermöglicht es dem Nutzer Listen mithilfe von wenigen Beispielen zu extrahieren und fortzuführen. Um Daten aus einer Liste zu extrahieren genügt es links oder rechts neben die Liste das gewünschte Ergebnis als Beispiel anzugeben. Folgende Beispiele sollen die Funktionsweise von Flash Fill verdeutlichen. Die Beispiel-Liste enthält den Deutschen WM-Kader von 2014 aus dem verschiedene Informationen extrahiert werden sollen.
Um Flash Fill zu benutzten reicht es eine Zelle mit Enter abzuschließen und in der nächsten Zelle weiterzuarbeiten . Man kann Flash Fill auch manuell unter Daten-Blitzvorschau im Reiter Datentools starten, siehe Abbildung 1

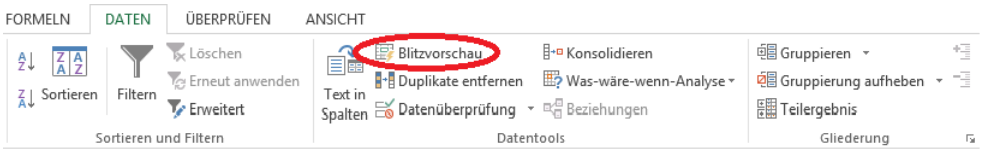

Abbildung 1: Reiter Datentools

Abbildung 2 zeigt wie man aus einer Liste den Vornamen bzw. den Nachnamen extrahieren und automatisch weiter auffüllen lassen kann. Dies erreicht man indem man eine neue Spalte erstellt und diese dann nur mit dem Vornamen/Nachnamen füllt. Flash Fill erkennt in diesem Beispiel nach nur einem Beispiel das Muster und der Nutzer kann dieses dann anwenden.

	A	B	C	D	E	F	G
1	Trikot	Vorname	Name	Geburtstag	Verein	Spiele	Tore
2	Tor						
3	1	Manuel	Manuel Neuer	27.03.1986	FC Bayern München	49	-
4	22	Roman	Roman Weidenfeller	06.08.1980	Borussia Dortmund	3	-
5	12	Ron	Ron-Robert Zieler	12.02.1989	Hannover 96	3	-
6	Abwehr						
7	20	Jérome	Jérome Boateng	03.09.1988	FC Bayern München	43	-
8	15	Erik	Erik Durm	12.05.1992	Borussia Dortmund	1	-
9	3	Matthias	Matthias Ginter	19.01.1994	SC Freiburg	2	-
10	2	Kevin	Kevin Großkreutz	19.07.1988	Borussia Dortmund	5	-
11	4	Benedikt	Benedikt Höwedes	29.02.1988	FC Schalke 04	25	2
12	5	Mats	Mats Hummels	16.12.1988	Borussia Dortmund	33	3
13	16	Philipp	Philipp Lahm	11.11.1983	FC Bayern München	110	5
14	17	Per	Per Mertesacker	29.09.1984	FC Arsenal	102	4
15	21	Shkodran	Shkodran Mustafi	17.04.1992	Sampdoria Genua	4	-
16	Mittelfeld						
17	14	Julian	Julian Draxler	20.09.1993	FC Schalke 04	11	1
18	19	Mario	Mario Götze	03.06.1992	FC Bayern München	33	10
19	6	Sami	Sami Khedira	04.04.1987	Real Madrid	49	4
20	23	Christoph	Christoph Kramer	19.02.1991	Borussia Mönchengladbach	3	-
21	18	Toni	Toni Kroos	04.01.1990	FC Bayern München	48	5
22	13	Thomas	Thomas Müller	13.09.1989	FC Bayern München	53	21
23	8	Mesut	Mesut Özil	15.10.1988	FC Arsenal	59	18

Abbildung 2: Vorname extrahieren

Abbildung 3 zeigt das Flash Fill auch den Inhalt einer Liste vertauschen kann. Aus der Spalte „Name“ werden die Namen entnommen und in der Spalte „Name Vertauschen“ in umgekehrter Reihenfolge wiedergegeben.

	A	B	C	D	E	F
1	Trikot	Vorname	Nachname	Name	Name Vertauschen	Geburtstag
2	Tor					
3	1	Manuel	Neuer	Manuel Neuer	Neuer Manuel	27.03.1986
4	22	Roman	Weidenfeller	Roman Weidenfeller	Weidenfeller Roman	06.08.1980
5	12	Ron	Zieler	Ron-Robert Zieler	Zieler Ron	12.02.1989
6	Abwehr					
7	20	Jérome	Boateng	Jérome Boateng	Boateng Jérome	03.09.1988
8	15	Erik	Durm	Erik Durm	Durm Erik	12.05.1992
9	3	Matthias	Ginter	Matthias Ginter	Ginter Matthias	19.01.1994
10	2	Kevin	Großkreutz	Kevin Großkreutz	Großkreutz Kevin	19.07.1988
11	4	Benedikt	Höwedes	Benedikt Höwedes	Höwedes Benedikt	29.02.1988
12	5	Mats	Hummels	Mats Hummels	Hummels Mats	16.12.1988
13	16	Philipp	Lahm	Philipp Lahm	Lahm Philipp	11.11.1983
14	17	Per	Mertesacker	Per Mertesacker	Mertesacker Per	29.09.1984
15	21	Shkodran	Mustafi	Shkodran Mustafi	Mustafi Shkodran	17.04.1992
16	Mittelfeld					
17	14	Julian	Draxler	Julian Draxler	Draxler Julian	20.09.1993
18	19	Mario	Götze	Mario Götze	Götze Mario	03.06.1992
19	6	Sami	Khedira	Sami Khedira	Khedira Sami	04.04.1987
20	23	Christoph	Kramer	Christoph Kramer	Kramer Christoph	19.02.1991
21	18	Toni	Kroos	Toni Kroos	Kroos Toni	04.01.1990
22	13	Thomas	Müller	Thomas Müller	Müller Thomas	13.09.1989
23	8	Mesut	Özil	Mesut Özil	Özil Mesut	15.10.1988

Abbildung 3: Namen vertauschen

Flash Fill setzt die String-Transformationen sehr gut um und erkennt auch sehr schnell Muster. Allerdings muss man diese Funktion erst einmal finden.

3 Grundlagen der Induktion funktionaler Programme

3.1 Funktionale Programmierung

Heutzutage gibt es eine Vielzahl unterschiedlicher Programmiersprachen. Zur besseren Übersicht kann man diese aufteilen in *imperative* und *deklarative* Programmiersprachen. Imperative Programmierung besteht, vergleichbar mit der grammatikalischen Bedeutung, aus einer Folge von Anweisungen, die nach der Reihe abgearbeitet werden. Im Gegensatz dazu steht die deklarative Programmierung, bei der lediglich das Problem mit allen wichtigen Sachverhalten und Zusammenhängen angegeben wird. Deklarative Programmiersprachen können weiter aufgegliedert werden, in einerseits Logiksprachen und andererseits funktionale Sprachen.
Die Grundlage für die Entwicklung funktionaler Programmierung schuf Alonso Church im Jahre 1936 durch seine Definition des funktionalen Systems *Lambda-Kalkül*. Die erste funktionale Programmiersprache ist LISP, die Anfang der 60er Jahre von John McCarthy entwickelt wurde um die Forschung in der künstlichen Intelligenz voranzutreiben. Obwohl funktionale Program-

mierung seit diesen Entwicklungen sehr geschätzt wurde, vor allem aufgrund der klaren Struktur, war es lange Zeit nicht effizient genug, funktionale Programme auf den vorhandenen Computern auszuführen. Diese verfügten über zu wenig Stackspeicher, der für die Ausführung von Rekursion essentiel ist. Heutige Rechner sind für diese Ausführung jedoch gut geeignet, hauptsächlich aufgrund der wesentlich größeren Speicher .
Funktionale Sprachen beruhen auf Mathematik und Logik. Sie bestehen aus einer Vielzahl an Funktionen ohne jeglicher Kontrollstrukturen. Funktionen haben die Eigenschaft, dass gleicher Input immer zu gleichem Output führt. Diese Beschaffenheit ist der Grund für das zustandslose Verhalten von funktionalen Programmiersprachen. Denn die Werte hängen nicht vom Zeitpunkt oder Zustand des Programms ab und sind deshalb immer gleich.
Eine wesentliche Kontrollstruktur in der funktionalen Programmierung ist die *Rekursion*. Auch wenn *Rekursion* und *Iteration* dazu führen, dass Aktionen wiederholt hintereinander ausgeführt werden, besteht doch ein großer Unterschied in der Durchführung. Bei der Iteration wird angegeben, dass eine Aktion mehrfach hintereinander ausgeführt wird. Häufig werden die Durchläufe dabei gezählt. Durch Rekursion jedoch ruft sich die Aktion immer wieder selbst auf. Dadurch wird das anfängliche Problem fortlaufend in kleinere Teilprobleme zerlegt, bis es schließlich gelöst ist. Rekursion ist auch, dass sich verschiedene Funktionen gegenseitig aufrufen: Funktion 1 ruft 2 auf, 2 ruft 3 auf und 3 ruft daraufhin wieder Funktion 1 auf. Es gibt eine Reihe von Aufgaben, die typischerweise durch Rekursion gelöst werden. Dazu zählt die Berechnung der Fakultät n!, die Lösung der Türme von Hanoi oder die Berechnung der Fibonacci-Zahlen. Die rekursive Berechnung der Fakultät wird im Folgenden erläutert.
Die Fakultät n! ist definiert als das Produkt der natürlichen Zahlen von 1 bis n. Einige Beispiele sind in der Tabelle 1 zu sehen.

n	1	2	3	4	5
$n!$	1	$1 \cdot 2$	$1 \cdot 2 \cdot 3$	$1 \cdot 2 \cdot 3 \cdot 4$	$1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$
$n!$	1	2	6	24	120

Tabelle 1: Beispiele zur Berechnung der Fakultät n!

Ein Sonderfall hierbei ist die Fakultät von 0, da $0! = 1$.
Eine generelle Funktion zur Berechnung von n! sieht folgendermaßen aus:
$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$
Um diese Funktion zur allgemeinen Berechnung zu programmieren, bietet sich eine rekursive Lösung an. Die Rekursion lässt sich im folgenden Beispiel

gut erkennen:

$4! = 4 \cdot 3 \cdot 2 \cdot 1$
$4! = 4 \cdot (3 \cdot 2 \cdot 1) \rightarrow 3!$
$4! = 4 \cdot 3!$

$3! = 3 \cdot 2 \cdot 1$
$3! = 3 \cdot (2 \cdot 1) \rightarrow 2!$
$3! = 3 \cdot 2!$

$2! = 2 \cdot 1$

$\rightarrow 4! = 4 \cdot (3 \cdot (2 \cdot 1))$

Bei jeder *Parameteränderung* findet Rekursion statt, also sobald ein Produkt durch eine Funktion, die Fakultät, ersetzt wird. Damit die Berechnung einer rekursiven Funktion terminieren kann, muss eine *Abbruchbedingung* vorhanden sein. In diesem Beispiel ist das der Fall, wenn 1! berechnet werden soll. Dieser Wert kann eindeutig als 1 definiert werden und dadurch die Berechnung beenden.
Die Abbruchbedingung (erste eq) und die Änderung der Parameter (zweite eq) werden in Maude (Pseudo-Code), wie folgt, sichtbar:

```
eq fac(1) = 1
eq fac(N) = N * (fac(N-1))
```

Um den Funktionen Wertemengen und den Wertemengen Eigenschaften zuzuweisen, benötigt man die Definition von *Datentypen*. Diese werden im nächsten Kapitel erläutert.

3.2 Alegebraische Datentypen

Essenziell für die Programmierung ist die Definition und Deklaration von Werten und Variablen. Dafür werden *Datentypen* verwendet. Datentypen bestehen aus einer Menge von Werten und der Menge von Operatoren über diese Wertemengen. Unterschieden werden dabei elementare Datentypen, wie zum Beispiel Integer, Boolean oder Char, aggregierte, abstrakte und algebraische Datentypen.
Angelehnt an das Vorlesungsskript „Funktionales Programmieren“ von Berthold Hoffmann[17], werden algebraische Datentypen hier weiter erläutert.
Der einfachste algebraische Datentyp ist der Aufzählungstyp. Dabei werden die Werte der Aufzählung durch den Datentyp der Werte eindeutig definiert. Im folgenden Beispiel werden die Werte des Datentyps Ziffer über

eine Aufzählung der Ziffern definiert.

```
type Ziffer = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

Durch *Pattern-matching*, also den Vergleich von Symbolen, können Funktionen auf diesen Aufzählungstyp gebildet werden. Ein Beispiel dafür ist die Überprüfung ob ein Wert im Datentyp enthalten ist mit der Ausgabe eines Booleans. Bsp:

```
isZiffer :: Ziffer → Bool
```

Ein weiterer algebraischer Datentyp ist der Produkttyp. Dabei werden Werte zu einem Datentyp zusammengefasst. Ein Tupel wird beispielsweise wie folgend definiert:

```
type Tupel = (Int,Int)
```

Damit wird erreicht, dass immer zwei Elemente zusammengehören. Summentypen sind Datentypen, die aus der Vereinigung von Wertemengen bestehen.
Algebraische Datentypen können weiterhin dadurch unterschieden werden, ob sie rekursiv sind oder nicht. Rekursive Datentypen können durch sich selbst definiert werden. Das bedeutet, dass sich der zu definierende Datentyp selbst beinhalten kann:

```
type Zahl = 0 | Add Zahl Ziffer
```

Bei diesem Beispiel wird der Datentyp *Zahl* durch die Addition einer Zahl und einer Ziffer definiert. Das Startelement 0 führt dazu, dass die Berechnung beendet werden kann. Ein weiteres Beispiel für rekursive Datentypen sind Bäume, da Knoten entweder leer sind oder einen Teilbaum beinhalten und dabei rekursiv aufgerufen werden.

3.3 Maude

Um die induktive Programmsynthese zu realisieren wurde das Tool Maude verwendet. Es handelt sich hierbei um ein „high-performance reflective language and system“, das vom Institut *SRI International* entwickelt wurde. Dieses System ermöglicht *Termersetzung* (term rewriting) und Anwendung von *Gleichungslogik*. Termersetzungsysteme sind Mengen von Termersetzungsregeln, welche vergleichbar mit Funktionen sind: f(a,b) → c. In der Gleichungslogik sind nur echte Gleichungen der Form a = b zulässig.
Die verwendete Version ist Maude 2, welche auf der Homepage der Entwick-

ler zu finden ist. Dort finden sich auch generelle Informationen, Dokumentationen und Weiteres zu dieser Thematik. [1] Um mit der Funktionsweise von Maude vertraut zu werden und das funktionale Programmieren besser zu verstehen, wurden zuerst einige einfache Beispiele programmiert, die im Folgenden erklärt werden. Auch auf die Syntax von Maude wird dabei weiter eingegangen. Zuerst betrachten wir die Funktion MyMember.

```
fmod MYMEMBER is
protecting NAT-LIST .
protecting BOOL .
op mem : Nat NatList -> Bool .

vars N M : Nat .
var L : NatList .

eq mem(N,nil) = false .
eq mem(N,(N L)) = true .
eq mem(N,(M L)) = mem(N,L) .
endfm
```

Bei dieser Funktion wird rekursiv ermittelt, ob ein bestimmter Wert in einer gegebenen Liste vorhanden ist. In der ersten Zeile leitet der Schlüsselterm fmod die Funktion ein, gefolgt von deren Name. protecting bedeutet, dass die angegebenen Datentypen geschützt werden und dadurch nicht überschrieben und verändert werden. Der Term `op mem` definiert den Funktionsoperator mit Namen mem und gibt dabei an, welche Datentypen verwendet werden: Eine natürliche Zahl und eine Liste von natürlichen Zahlen sind der Input und als Output bekommt man einen Boolean-Wert. In der nächsten Zeile werden die verwendeten Variablen ihren Datentypen zugeordnet. Nun folgen die eigentlichen Funktionsregeln. Eq gibt dabei an, dass es sich um equations, also Gleichungen handelt. Die erste Gleichung sagt aus, dass wenn nach N gesucht ist und die gegebene Liste leer ist, der Ausgabewert false ist. In der zweiten Gleichung wird angegeben, dass falls das gesuchte N am Anfang der Liste steht, true ausgegeben wird. In der 3. Gleichung ist die Rekursion zu finden. Falls das gesuchte N nicht am Anfang der Liste steht, wird das erste Element entfernt und die Funktion erneut auf die Restliste angewendet. Dies wiederholt sich so lange, bis einer der ersten beiden Fälle eintritt und eine Lösung gefunden werden kann. Der Schlüsselterm endfm schließt die Funktion ab.

```
fmod MYLAST is
protecting NAT-LIST .
op last : NatList -> Nat .
```

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

```
var N : Nat .
vars L X Y : NatList .

eq last(N nil) = N .
eq last(X (N nil)) = last(N nil) .
endfm
```

Im Folgenden wird nur noch auf die Funktionsweise der Maude-Funktionen eingegangen und die Syntax beiseite gelassen.
Die Funktion MYLAST gibt den letzten Wert einer Liste aus und arbeitet auf ganz ähnliche Weise, wie MYMEMBER. Falls eine Liste nur ein Element beinhaltet, wird dieses ausgegeben. Falls eine Liste mehr als ein Element enthält, wird der erste Wert entfernt und die Funktion rekursiv auf die Restliste angewandt bis das letzte Element gefunden wurde.
Um Funktionen in Maude auszuführen, muss die Funktion in einer Textdatei mit der Endung .maude gespeichert sein. Diese wird dann mithilfe des Befehls load in Maude geladen. Danach kann die Funktion ausgeführt werden, wie im folgenden Beispiel gezeigt wird.

```
red in NAT-LIST : last(1 2 3 4) .
```

Hierbei leitet der Term `red` die Reduktion, also Berechnung ein. Der Ausdruck `in ...` gibt an, um welchen Datentyp es sich handelt. Nach einem Doppelpunkt folgt dann der Name des gewünschten Operators und die Liste, auf die die Funktion angewendet werden soll. Die komplette Durchführung ist in Abbildung 4 zu sehen.

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
           Maude 2.6 built: Dec 10 2010 11:12:39
           Copyright 1997-2010 SRI International
                   Tue Jul  8 19:53:10 2014
Maude> load mylast.maude
Maude> red in NAT-LIST : last(1 2 3 4) .
reduce in NAT-LIST : last(1 2 3 4) .
rewrites: 1 in 0ms cpu (0ms real) (32258 rewrites/second)
result NzNat: 4
Maude>
```

Abbildung 4: MyLast in Maude

3.4 Igor

Das System, das zur Induktiven Programmsynthese benutzt wird heißt IGOR. Die verwendete Version davon ist IGOR 2.2, welche an Maude angepasst

wurde. Igor wendet das Prinzip der Termersetzung und des pattern matching an, um rekursive funktionale Programme zu generieren. Dies bedeutet, dass das System Gesetzmäßigkeiten in Input/Output-Beispielen findet und daraus Termersetzungsregeln erstellt. Diese Regeln sind dann die Grundlage für funktionale Programme.
Um IGOR verwenden zu können, muss zuerst Maude funktionieren. Nachdem Maude gestartet wurde, kann die Datei igor2.2.maude mit dem Begriff load eingelesen werden. Daraufhin wird auf die gleiche Weise ein Igormodul eingelesen. Ein Beispiel für ein solches Modul wird im Folgenden erklärt. Dabei wird auch auf die Syntax, die sich von der von Maude kaum unterscheidet, eingegangen.

```
fmod LAST is
sorts MyItem MyList InVec .

op # : -> MyList [ctor] .
op cons : MyItem MyList -> MyList [ctor] .
op last : MyList -> MyItem [metadata "induce"] .
op in : MyList -> InVec [ctor] .

vars U V W  : MyItem .

eq last(cons(U,#)) = U .
eq last(cons(U,cons(V,#))) = V .
eq last(cons(U,cons(V,cons(W,#)))) = W .

endfm
```

Hierbei handelt es sich um ein Modul, bei dem aus einer Liste das letzte Element ermittelt werden soll. Zu Beginn wird der Name des Moduls genannt. Durch `sorts` werden die Datentypen genannt, die benutzt werden. Dabei handelt es sich um spezielle Datentypen, die in Igor definiert sind. Daraufhin werden die verwendeten Operatoren definiert. Bei # handelt es sich in diesem Beispiel um ein Symbol, das für das leere Element steht. Der Operator `in` wird von Igor benötigt, um Werte in seinen eigenen Datentyp Invec zu transformieren. Sobald der Operator der Funktion definiert wird, die Igor lernen soll, ist es notwendig [metadata „induce“] an das Ende der Zeile zu hängen, damit Igor weiß, dass diese Funktion generiert werden soll. Danach werden die Variablen ihren Datentypen zugeordnet. Nun folgen die Input/Output-Beispiele in Form von Gleichungen. Das sind die Daten, in denen Igor Regularitäten erkennt und aus denen die Termersetzungsregeln generiert werden.

Damit Igor diese Funktion lernen kann, muss, nach dem einladen des Moduls, folgender allgemeiner Befehl eingegeben werden:
`reduce in IGOR : gen('MODULNAME, 'FUNKTIONSNAME, 'BKQ) .`
An die Stelle des BKQ, also des Backgroundknowlegde, kommen, falls nötig, Funktionen als Hintergrundwissen. Diese werden in der Form `'func1 * 'func2 * 'funcn` angegeben. Falls kein Hintergrundwissen notwendig ist wir der Begriff `noName` anstelle von 'BKQ geschrieben. Um das zuvor erläuterte Beispiel starten zu können lautet der Befehl wie folgt:
`'reduce in IGOR : gen('LAST, 'last, noName) .`
Wurde alles korrekt durchgeführt, generiert Igor nun folgende Ausgabe:

```
result Hypo: hypo(true, 2, eq 'last['cons['X0:MyItem,'nil.MyList]]
 = 'X0:MyItem[none] .
 eq 'last['cons['X0:MyItem,'cons['X1:MyItem,'X2:MyList]]]
 = 'last['cons['X1:MyItem,'X2:MyList]] [none] .)
```

Es handelt sich hierbei um eine sogenannte Hypolist, eine Liste bestehend aus mindestens einer Hypothese, welche wiederum aus mindestens einer Equation besteht. Die Quintessenz der Ausgabe findet sich in den Gleichungen, welche mit `eq` eingeleitet werden. Die Werte werden dabei als X mit fortlaufender Nummerierung benannt. Außerdem wird der Datentyp der Variablen mitgeteilt. Bei dem vorigen Beispiel lautet die erste Gleichung:

```
eq 'last['cons['X0:MyItem,'#.MyList]]= 'X0:MyItem .
```

Diese sagt aus, dass die Funktion `last` bei einer einelementigen Liste mit leerer Restliste (`#`) das einzige Element zur Lösung hat. Hierbei handelt es sich um die Abbruchbedingung. Die zweite Gleichung lautet:

```
eq 'last['cons['X0:MyItem,'cons['X1:MyItem,'X2:MyList]]] =
'last['cons['X1:MyItem,'X2:MyList]] .
```

Diese Gleichung sagt aus, dass bei einer Liste mit zwei Werten und Restliste das erste Element entfernt wird und die Funktion `last` auf die Liste mit dem zweiten Element und der Restliste angewandt wird. Hierbei wird die Änderung der Parameter und die rekursive Lösung sichtbar. Insgesamt lautet die Vorgehensweise zur Lösung: Entferne immer das erste Element einer Liste und wende `last` auf die Restliste an bis nur noch ein Element übrig ist. Dieses ist dann das gesuchte Element. Genau diese Funktionsweise führt zum letzten Wert einer Liste und sollte induziert werden.
Die komplette Durchführung wird in der Abbildung 5 gezeigt:

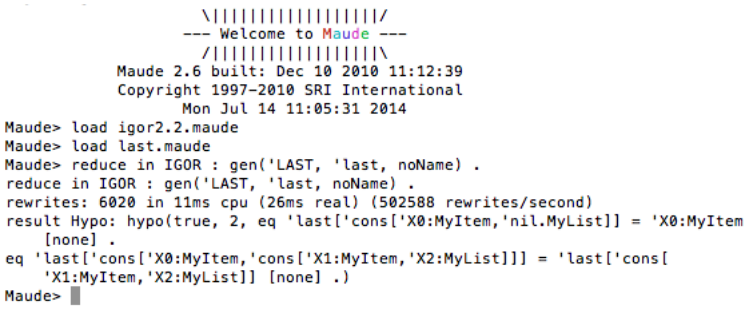

Abbildung 5: Last in Igor

4 Benutzeroberfläche für Igor und Maude

4.1 Einführung und Anforderungen

Um die Bedienung von IGOR und Maude zu vereinfachen, wurde im Verlauf des Projekts eine Benutzeroberfläche dafür entwickelt. Diese sollte einfach zu benutzen sein und alle nötigen Schritte automatisiert ausführen. Sie vereint alle für die Bedienung von IGOR notwendigen Elemente an zentraler Stelle. Die Oberfläche kann Beispieldateien laden, anschließend kann das gewünschte Beispiel ausgewählt werden. Hiernach kann IGOR gestartet und die Hypothesen für dieses Beispiel erzeugt werden, welche automatisch in eine für den Anwender besser lesbare Ausgabe umgewandelt werden. Zudem erzeugt das Programm in diesem Schritt noch validen Maudecode, der in das Beispielprogramm eingefügt und getestet werden kann. Letzteres ist zum Zeitpunkt der Erstellung dieses Dokuments noch nicht vollständig implementiert, der generierte Programmtext wird aber in die Ergebnisausgabe mit eingefügt.

4.2 Implementierung

Realisiert und implementiert wurde die Benutzeroberfläche in Java. Alle GUI-Elemente wurden mit Swing realisiert, die darunterliegende Hauptlogik des Programms wurde als Model-View-Controller mit Observer/Observable-Pattern umgesetzt. Die Implementierung der integrierten Parser, die die IGOR-Ausgabe verarbeiten und hieraus wieder Maudecode generieren, erfolgte mit Hilfe der Parserbibliothek *parboiled*. Diese stellt das Grundgerüst für die verwendeten Parser bereit, wird hier aber nicht näher erläutert, da dies den Rahmen sprengen würde.

4.3 Bedienung

Nach dem Start des Programms wird dem Benutzer das Hauptfenster der Anwendung gezeigt, über welches alle weiteren Aktionen gestartet werden

können. Dies ist in Abbildung 6 zu sehen. An ihr sollen auch alle weiteren Schritte, die zur Benutzung des Programms nötig sind, erläutert werden.

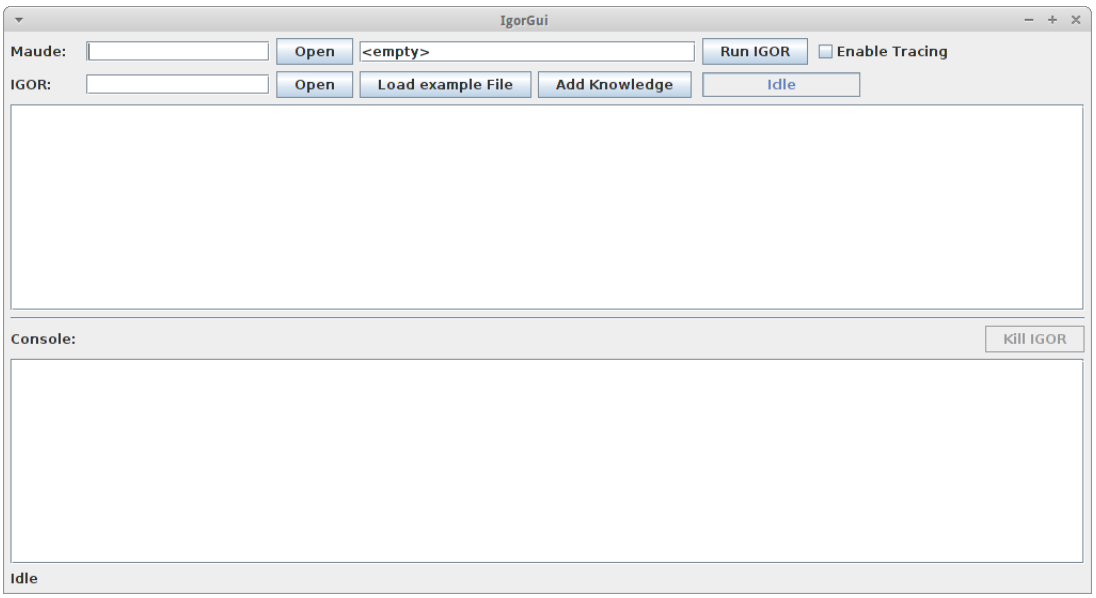

Abbildung 6: IgorGUI direkt nach Programmstart

Im oberen drittel des Fensters findet sich der Einstellungsabschnitt. Hier müssen zuerst die Pfade für Maude und IGOR angegeben werden, auf die das Programm im weiteren Verlauf zurückgreift. Der Benutzer kann die Pfade entweder direkt in die entsprechenden Textfelder eintragen, oder über die jeweilige *Open*-Schaltfläche die entsprechenden Dateien im Dateisystem suchen und auswählen. Die Auswahldialoge filtern hierbei nach den entsprechenden Dateitypen, dies sind für das Maude-Feld ausführbare/binäre Dateien und für IGOR `.maude`-Dateien.
Als Nächstes sollte der Benutzer über *Load example File* eine Maudedatei mit entsprechenden Beispielen laden, die IGOR verarbeiten soll. Nachdem die selektierte Datei verarbeitet wurde kann der Benutzer über die Auswahl über der Schaltfläche eines der gefundenen Beispiele zur weiteren Verarbeitung markieren. Dessen Programmcode wird dann im Textfeld unter den Bedienelementen angezeigt. Auf Wunsch kann über die zweite Schaltfläche in diesem Bereich, *Add Knowledge*, noch zusätzliches Vorwissen für IGOR angegeben werden. Der entsprechende Dialog gibt hierzu eine eigene, kurze Anleitung, wie in Abbildung 7 gezeigt.

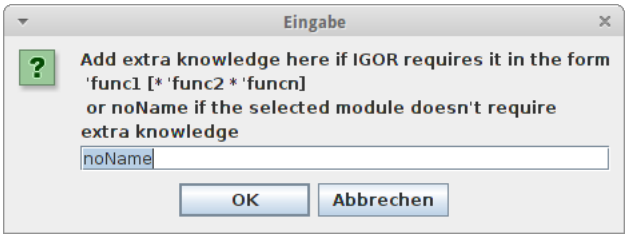

Abbildung 7: Dialog Add Knowledge

Nachdem der Benutzer alle Einstellungen vorgenommen hat kann die eigentliche Ausführung von IGOR über die Schaltfläche *Run IGOR* gestartet werden. In diesem Zustand sieht das Programmfenster wie in Abbildung 8 aus.

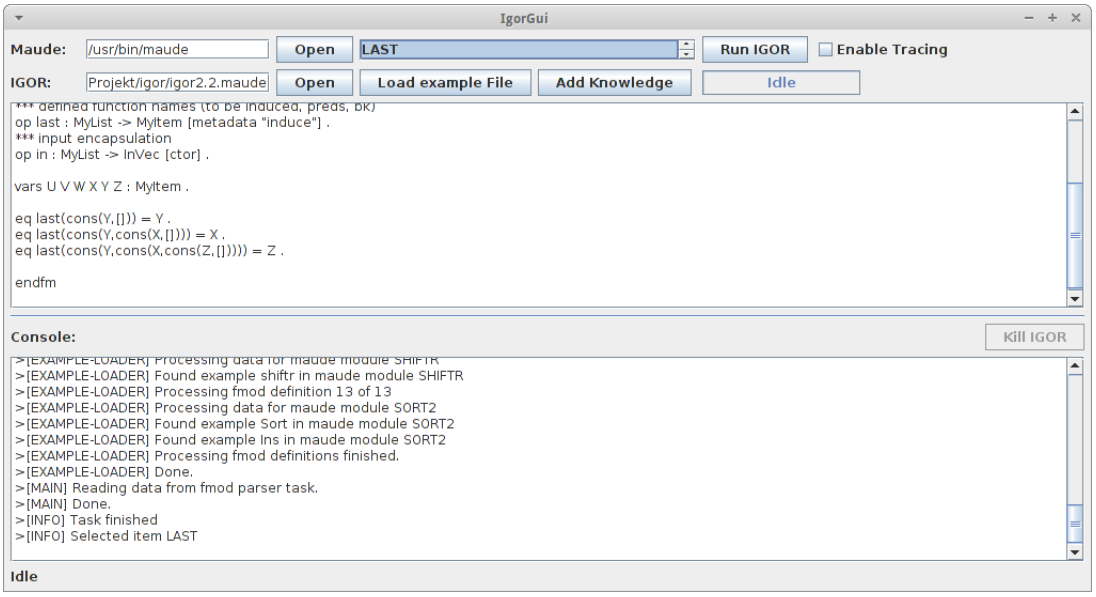

Abbildung 8: IgorGUI mit korrekten Einstellungen und ausgewähltem Beispiel

Stellt das Programm beim Start von IGOR fest, dass das ausgewählte Beispiel mehrere Funktionsdefinitionen enthält, so wird dem Benutzer wie in Abbildung 9 dargestellt eine Auswahlmöglichkeit angezeigt.

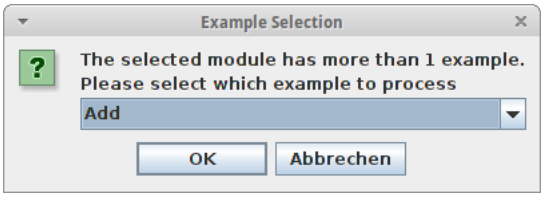

Abbildung 9: Auswahldialog bei Beispiel mit mehreren Funktionen

Ebenso wird dem Benutzer eine Warnung angezeigt, sollte die Checkbox *Enable Tracing* aktiviert sein (siehe Abbildung 10). Dies aktiviert den in Maude integrierten Tracebefehl, der sämtliche Arbeitsschritte eines Maudeprogramms ausgibt. Im Fall von IGOR erzeugt dies einiges an Ausgabe, welche intern gepuffert und für die anschließende Weiterverarbeitung zwischengespeichert wird, ehe sie im zweiten Textfeld der Benutzeroberfläche ausgegeben wird. Dies kann unter Umständen dazu führen, dass die Benutzeroberfläche langsam reagiert oder - in extremen Fällen - abstürzt. Falls der Benutzer in diesem Dialog auf *Ja* klickt, bleibt Tracing aktiviert, ansonsten wird es wieder deaktiviert. Anschließend wird der IGOR-Prozess gestartet.

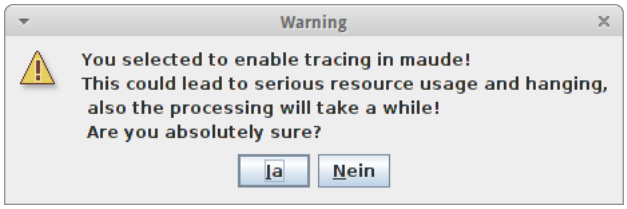

Abbildung 10: Warnung bei aktiviertem Tracing

Nachdem alle Einstellungen getätigt wurden und das Programm arbeitet zeigt sich das Fenster wie in Abbildung 11. Die Prozessleiste im oberen Drittel der Oberfläche bewegt sich, ebenso werden in der Fußzeile des Programmfensters Informationen zur Speicherauslastung angezeigt, wenn der IGOR-Prozess länger als zwei Sekunden arbeitet (zum Beispiel bei aktiviertem Tracing). Ebenso werden sämtliche Ausgaben des IGOR-Prozesses und Fehlermeldungen im Konsolen-Textbereich ausgegeben. In diesem Zustand kann der Benutzer kein weiteres Beispiel von IGOR verarbeiten lassen. Dies ist aus Gründen der Ressourcennutzung Absicht und wird im nächsten Kapitel noch genauer erläutert.

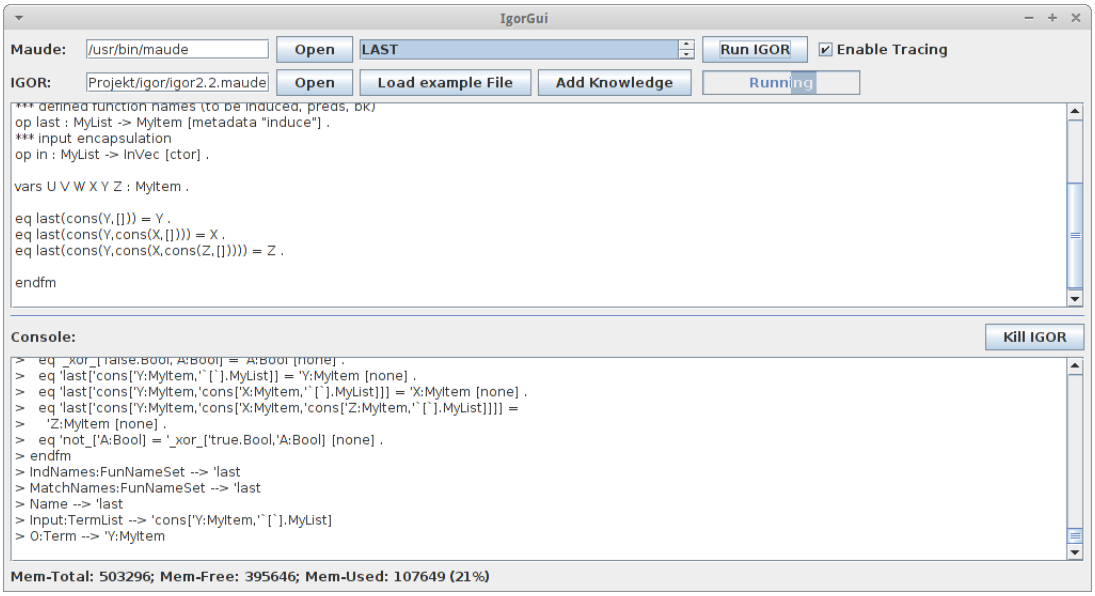

Abbildung 11: IgorGUI während IGOR-Durchlauf

Wenn IGOR schließlich zu einer Lösung gekommen ist werden die Hypothesen daraus im Konsolen-Textbereich ausgegeben, ebenso generiert das Programm für jede Hypothese validen Maudecode, der wiederum den von IGOR benutzten Funktionscode im selektierten Beispiel ersetzen soll, um ein funktionierendes, rekursives Maudeprogramm zu erhalten. Es war zudem geplant, Letzteres zu implementieren, um das Testen der generierten Hypothesen zu vereinfachen. Aufgrund der begrenzten zeitlichen Ressourcen für dieses Projekt wurde davon jedoch abgesehen. Die generierte Ausgabe ist in Abbildung 12 zu sehen. In diesem Zustand kann man mit dem Programm nun ein neues Beispiel aus der Liste auswählen oder eine neue Beispieldatei laden und den Prozess von Vorne beginnen. Der aktuell laufende Prozess kann jederzeit über die Schaltfläche *Kill IGOR* die sich über dem Konsolentextfeld befindet angehalten werden.

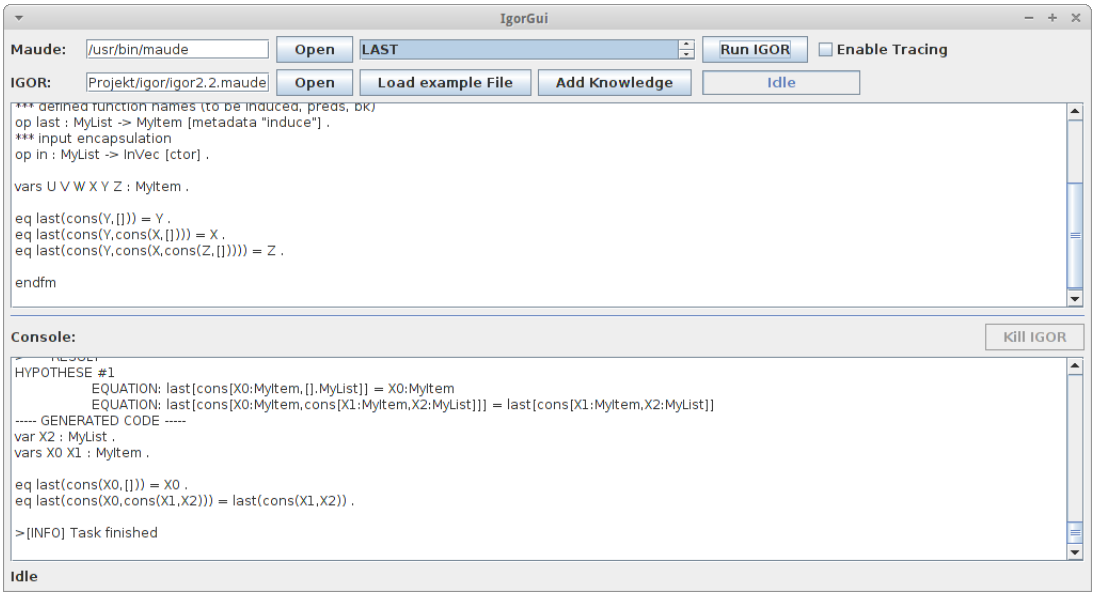

Abbildung 12: IgorGUI nach Beenden eines Durchlaufs

Damit wäre nun eine kurze Anleitung über die IGOR-Bedienoberfläche gegeben. Im nächsten Kapitel folgt ein Überblick über bekannte Probleme und Restriktionen des Programms.

4.4 Bekannte Probleme und Restriktionen

Zuerst sei gesagt, dass der Fokus der Entwicklung angesichts des begrenzten Zeitrahmens des Projekts auf Funktionalität als auf Stabilität lag.
Natürlich werden die gröbsten Fehler, die sich im Programmablauf ergeben können, ordnungsgemäß abgefangen. Dennoch können unbeabsichtigt Fehlerfälle auftreten, die sich aus den recht komplexen Interna ergeben. Ebenso stellt die Ressourcenverwaltung ein Problem dar, insbesondere im Hinblick auf den verwendeten Speicher. Im Ernstfall kann dies dazu führen, dass das Programm sich aufhängt und neu gestartet werden muss. Dies zeigt sich insbesondere bei längeren Durchläufen von IGOR, hierbei alloziert das Programm immer mehr Speicher, bis ein stabiler Betrieb nicht mehr garantiert ist. Hier herrscht noch Potential zur Verbesserung, ebenso muss der während der Entwicklung „gewachsene“ Programmcode an vielen Stellen noch optimiert werden.
Aufgrund des internen Aufbaus und der intensiven Speichernutzung kann der Benutzer keine zwei oder mehr IGOR-Prozesse gleichzeitig ausführen. Es wurde versucht, dies zu implementieren, scheiterte aber an den bereits beschriebenen Speicherproblemen und der benötigten Synchronisation zwischen den Instanzen, die Maude und IGOR ausführen, und des Prozesses,

der für die Konsolenausgabe im Hauptfenster zuständig ist. Diese wäre bei mehreren gleichzeitig laufenden Prozessen recht komplex und damit fehleranfällig geworden, deshalb wurde der Einfachheit halber darauf verzichtet.
Viele Probleme den Speicher betreffend lassen sich vermeiden oder zumindest reduzieren, indem man der Virtuellen Maschine, in der das Programm ausgeführt wird, entsprechend mehr Speicher zuweist, zum Beispiel über die JVM-Argumente `-Xms512m -Xmx512m -XX:PermSize=256m`. Dies weißt dem Programm 512 MB Hauptspeicher fest zu und erlaubt eine maximale JVM-Stackgröße von 256 MB. Hierzu sei gesagt, dass das Programm zur Ausführung zwingend Java in der Version 1.7 benötigt, da intern viele Techniken Verwendung finden, die in früheren Java-Versionen nicht verfügbar sind.
Da Java eine betriebssystemunabhängige Programmiersprache ist sollte das Programm sowohl unter Windows als auch unter Linux und anderen unixoiden Betriebssystemen laufen. Intensiv getestet wurde das Programm allerdings nur unter Ubuntu 14.04, da es mit der Installation von Maude unter Windows Probleme gab.

4.5 Lizenzen

Als Abschluss dieses Kapitels erfolgt noch ein knapper Überblick über die Lizenzen von IgorGUI selbst und der verwendeten externen Bibliotheken.
Sämtlicher Quellcode der im Rahmen dieses Projekts verfasst wurde steht unter der *GNU General Public License* in Version 3. Dies erlaubt die Weitergabe und Modifikation des Quelltextes auch nach Abschluss dieses Projekts.
Die verwendete Parserbibliothek *parboiled* steht unter der *Apache License Version 2.0* und darf daher ebenfalls ohne Einschränkungen weitergegeben werden, unter Beachtung der Lizenzkriterien. Es ergeben sich allerdings keine nennenswerten Einschränkungen hieraus, da die Apache-Lizenz in Version 2 und die GNU GPL in Version 3 in der Hinsicht kompatibel zueinander sind, als dass Projekte unter GNU GPL Version 3 Projekte und Programme bzw. Programmteile unter Apache-Lizenz in Version 2 enthalten dürfen.
Die letzte verwendete Bibliothek, MigLayout, ist für den Aufbau der Swing-Benutzeroberfläche zuständig und steht vollständig unter BSD- bzw. GPL-Lizenz und darf ohne Einschränkungen weitergegeben werden.
Sämtlicher Quellcode und die benötigten Zusatzbibliotheken stehen im Git-Repository des Autors[2] zum Abruf bereit.

[2] `https://bitbucket.org/thepete89/igorgui`

5 Evaluation von Flash Fill-Beispielen mit Igor

5.1 Liste zu Liste

Unser erster Ansatz war es aus einer Liste mit Strings eine neue Liste zu erstellen, welche für Vornamen den ersten, dritten, fünften, usw. und für Nachnamen den zweiten, vierten, sechsten, usw. String der Ursprungsliste enthalten soll.
Für dieses Vorgehen benötigen wir die Datentypen MyString und MyList. Für MyList definieren wir folgende Operatoren:

```
op nil : -> MyList [ctor] . - Die leere Liste
op cons : MyString MyList -> MyList [ctor] . - Aus MyString
und einer MyList wird eine MyList
```

Unser Hauptoperator heißt „firstname“:

```
op firstname : MyList -> MyList [metadata "induce"] .
```

Dieser Operator macht aus einer MyList eine MyList.
Des Weiteren werden einige Variablen vom Typ MyString benötigt:

```
vars X Y Z U V W : MyString .
```

Die Equations für Vornamen:

```
eq firstname(cons(X,nil)) = cons(X,nil) .
eq firstname(cons(X,cons(Y,nil))) = cons(X,nil) .
eq firstname(cons(X,cons(Y,cons(Z,nil)))) = cons(X,cons(Z,nil)) .
eq firstname(cons(X,cons(Y,cons(Z,cons(U,nil))))) =
cons(X,cons(Z,nil)) .
eq firstname(cons(X,cons(Y,cons(Z,cons(U,cons(V,nil)))))) =
cons(X,cons(Z,cons(V,nil))) .
eq firstname(cons(X,cons(Y,cons(Z,cons(U,cons(V,cons(W,nil))))))) =
cons(X,cons(Z,cons(V,nil))) .
```

Die erste Equation benutzt „firstname“ welches eine MyList erwartet. Diese MyList wird mit „cons“ erstellt und enthält den MyString „X“ und eine leere Liste. Das Ergebnis ist dann wieder eine MyList bestehend aus „cons“ mit den Elementen „X“ und die leere Liste. Dies bedeutet man erstellt aus einer MyList die nur das Element „X“ enthält eine Liste mit den Element „X“.
Die folgenden Equations arbeiten nach dem selben Prinzip, jedoch enthalten die MyList mehrere Elemente.
Aus den Equations lernt IGOR nun aus einer Liste mit Strings gezielt Strings zu extrahieren und diese in eine neue Liste zu schreiben.

Das vollständige Programm sieht folgendermaßen aus:

```
fmod FIRSTNAME is
sorts MyString MyList InVec .

op [] : -> MyList [ctor] .
op cons : MyString MyList -> MyList [ctor] .
op in : MyList -> InVec [ctor] .

op firstname : MyList -> MyList [metadata "induce"] .

vars X Y Z U V W : MyString .

eq firstname(cons(X,nil)) = cons(X,nil) .
eq firstname(cons(X,cons(Y,nil))) = cons(X,nil) .
eq firstname(cons(X,cons(Y,cons(Z,nil)))) = cons(X,cons(Z,nil)) .
eq firstname(cons(X,cons(Y,cons(Z,cons(U,nil))))) =
cons(X,cons(Z,nil)) .
eq firstname(cons(X,cons(Y,cons(Z,cons(U,cons(V,nil)))))) =
cons(X,cons(Z,cons(V,nil))) .
eq firstname(cons(X,cons(Y,cons(Z,cons(U,cons(V,cons(W,nil))))))) =
cons(X,cons(Z,cons(V,nil))) .

endfm
```

5.2 Erster Buchstabe

Um den Anfangsbuchstaben aus einem String zu extrahieren benötigen wir die Datentypen MyString und MyChar. Für MyString definieren wir:

```
op "" : -> MyString [ctor] .  - leerer String
```

Und für MyChar:

```
op # : -> MyChar [ctor] .  - leerer Char
op ins : MyChar MyString -> MyString [ctor] .  - MyChar und
MyString ergeben einen MyString
```

Des Weiteren definierten wir folgende Operatoren, die für dieses Programm nicht gebraucht werden, jedoch bei einer möglichen Fortsetzung der Arbeit hilfreich seien könnten.

```
op tostring : MyChar -> MyString [ctor] .
op + : MyString MyString -> MyString [ctor] .
```

Der Operator „firstletter“ führt die Hauptaufgabe des Programms durch indem er aus einem MyString ein MyChar macht.

```
op firstletter : MyString -> MyChar [metadata "induce"] .
```

Die Variablen für diese Aufgabe sind ein MyString und ein MyChar.

```
var S : MyString .
var A : MyChar .
```

Die benötigten Equations:

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

Die erste Equation erzeugt aus einen leeren MyString ein leeres MyChar. Die zweite Equation erhält in „firstletter“ „ins(A,)“, einen MyString bestehend aus „A“ und dem leeren MyString und erzeugt daraus den MyChar „A“. Die dritte Equation funktioniert wie die Zweite, erhält in „ins“ allerdings keinen leeren MyString.

Das vollständige Programm:

```
fmod FIRSTLETTER is
sorts MyString MyChar InVec .

***op tostring : MyChar -> MyString [ctor] .
op "" : -> MyString [ctor] .
op # : -> MyChar [ctor] .
***op + : MyString MyString -> MyString [ctor] .
op ins : MyChar MyString -> MyString [ctor] .
op in : MyString -> InVec [ctor] .
op firstletter : MyString -> MyChar [metadata "induce"] .

var S : MyString . var A : MyChar .

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

endfm
```

5.3 Liste mit Tupeln

Dieser Ansatz geht davon aus, dass Namen als Tupel zu sehen sind und nimmt daher eine Liste mit Tupel und erstellt daraus eine Liste mit Strings. Wir gehen davon aus, dass die Tupel die Form (Vorname, Nachname) haben.

Hierfür benötigen wir neben den bereits bekannten Datentypen MyString und MyList noch den Datentyp MyTupel. Für MyTupel benötigen wir diesen Operator

```
op tup : MyString MyString -> MyTupel [ctor] .
```

Um das Ergebnis in die richtige Form zu bekommen verwenden wir einen weiteren Operator. Dieser erstellt aus einem MyString und einer MyList wieder eine MyList.

```
op result : MyString MyList -> MyList [ctor] .
```

Die restlichen Operatoren wurden an die neuen Datentypen angepasst.

```
op # : -> MyString [ctor] .
op nil : -> MyList [ctor] .
op tup : MyString MyString -> MyTupel [ctor] .
op cons : MyTupel MyList -> MyList [ctor] .
```

Der Operator „firstname“ aus dem ersten Ansatz bleibt gleich wurde jedoch in „fname“ umbenannt. Für diesen Ansatz benötigen wir mehrere Variablen die alle vom Typ MyString sind.

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

Die Equations für diesen Ansatz sind umfangreicher als bei den vorherigen Versuchen.

```
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)))))) .
```

Wir werden hier die erste Equation erklären, da die restlichen dem gleichem Muster folgen. Wir übergeben „fname“ wieder ein „cons“, dieses enthält dieses Mal jedoch nicht mehr einen MyString und eine MyList, sondern einen MyTupel und eine MyList. Dieses MyTupel besteht aus den beiden MyStrings „S“ und „T“. Das Ergebnis ist nun eine MyList die mithilfe von „result“ erstellt wird. Diese MyList enthält nun den MyString „S“ und eine leere MyList, d.h. das erste Element des MyTupels und somit den Vornamen. Die restlichen Equations enthalten Beispiele für mehrere MyTupel in einer Liste.
Unser vollständiges Programm:

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

op # : -> MyString [ctor] .
op nil : -> MyList [ctor] .
op tup : MyString MyString -> MyTupel [ctor] .
op cons : MyTupel MyList -> 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)))))) .

endfm
```

6 Konklusion

Das Ziel dieser Arbeit war es, an Excels Flash-Fill angelehnte Stringtransformationen mit dem allgemeinen Tool Igor umzusetzen. Dazu wurden zuerst die dafür notwendigen Grundlagen, wie zum Beispiel die Funktionsweisen von Maude und Igor, ermittelt und erläutert. Des weiteren wurde eine Benutzeroberfläche zur einfacheren Handhabung der beiden genannten Tools entwickelt. Als Kern dieser Arbeit wurden daraufhin Input/Output-Beispiele ermittelt, die Igor benötigt, um die Transformationen zu lernen. Diese wurden dann in entsprechende Module überführt, damit sie in Igor durchgeführt, also gelernt werden konnten.
Eine der Schwierigkeiten bei der Durchführung dieser Forschungsarbeit war zuerst ein Verständnis für die, für uns neue, funktionale Programmierung zu bekommen. Außerdem fiel es zuerst nicht leicht, mit der spärlichen Feh-

leranzeige in der Konsole, die Funktionsweisen von Maude und Igor zu verstehen. Doch trotz, oder gerade aufgrund, dieser Hindernisse konnte diese Forschungsarbeit erfolgreich zum Abschluss gebracht werden.
Diese Arbeit zeigt, dass Igor die gewählten Flash-Fill Beispiele erlernen kann. Nach unserer Meinung ist Igor durchaus in der Lage mithilfe geeigneter Input/Output-Beispiele auch die restlichen Flash-Fill Funktionen umsetzten zu können. Weiterhin gehen wir davon aus, dass mit Igor Transformationen gelernt werden können, die nicht zu Flash-Fills Möglichkeiten gehören. Dieser Ansatz könnte eine weitere Fragestellung sein, die hoffentlich in der Zukunft erforscht wird.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *Apache License Version 2.0*. The Apache Software Foundation. http://www.apache.org/licenses/LICENSE-2.0.html

[2] *Gabler Wirtschaftslexikon: Definition Datentyp*. Springer Gabler Verlag. http://wirtschaftslexikon.gabler.de/Definition/datentyp.html

[3] *Gabler Wirtschaftslexikon: Definition Programmiersprache*. Springer Gabler Verlag. http://wirtschaftslexikon.gabler.de/Definition/programmiersprache.html?referenceKeywordName=deklarative+Programmiersprache

[4] *Igor Maude Readme*. http://www.cogsys.wiai.uni-bamberg.de/IgorMaude/README.txt

[5] *Imperative und funktionale Sprachen*. http://www.betoerend.de/dasLandHinterDemEndeDesSinns/lambda/folien7bis9.html

[6] *Maude language and tool*. NTNU Trondheim. http://www.ntnu.no/telematikk/department/labs/topics/maude

[7] *Maude Overview*. SRI International. http://maude.cs.uiuc.edu/overview.html

[8] *Parboiled Wiki Introduction*. Parboiled Team. https://github.com/sirthias/parboiled/wiki

[9] *THE GNU GENERAL PUBLIC LICENSE VERSION 3*. Free Software Foundation. http://www.gnu.org/licenses/gpl-3.0.en.html

[10] *Was ist funktionale Programmierung?* http://www.infosun.fim.uni-passau.de/cl/lehre/funcprog05/wasistfp.html

[11] Alp, A. ; Uzun, Y. ; Alkan, N.: *Funktionale Programmierung*. http://www.deinprogramm.de/MUPIS/Uberblick_files/funktional.pdf

[12] Böhme, P.: *Arten von Datentypen*. http://www2.informatik.uni-halle.de/lehre/pascal/sprache/datentyp.html

[13] Dianhuan, L. ; Dechter, E. ; Ellis, K. ; Tenenbaum, J. B. ; Muggleton, S. H.: *Bias reformulation for one-shot function induction*. – submitted

[14] Ehses, E.: *Paradigmen der Programmierung - Funktionale Programmierung*. http://www.gm.fh-koeln.de/ehses/paradigmen/folien/scala.pdf

[15] Grimm, R.: *Funktionale Programmierung(1): Grundzüge.* http://www.linux-magazin.de/Online-Artikel/Funktionale-Programmierung-1-Grundzuege

[16] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: *POPL* (2011)

[17] Hoffmann, B.: *Algebraische Datentypen.* http://www.informatik.uni-bremen.de/agbkb/lehre/pi3/folien/Algebraisch.pdf

[18] Hofmann, M.: *Automated Construction of XSL-Templates - An Inductive Programming Approach.* http://www.cogsys.wiai.uni-bamberg.de/theses/hofmann/hofmann.pdf, Otto-Friedrich-Universität Bamberg, Diplomarbeit, 2007

[19] Kitzelmann, E.: *Inductive Programming - Ein Überblick und der IGOR II-Algorithmus.* http://www.cogsys.wiai.uni-bamberg.de/effalip/data/talks/bremen08.pdf. Version: 2008

[20] Rohne, G.: *Igor II Readme.* http://www.iai.uni-bonn.de/~grohne/darcs/igor2/README

[21] Schmitt, P. H.: *Formale Systeme: Gleichungslogik.* http://i12www.ira.uka.de/~pschmitt/FormSys/Folien0506/21Glogik.pdf

[22] Sneal, C.: *Maude Course Chapter 1: Term Rewriting.* http://www.cs.swan.ac.uk/~csneal/MaudeCourse/termrewriting.html

[23] Sneal, C.: *Maude Course Chapter 4: Basic Maude.* http://www.cs.swan.ac.uk/~csneal/SystemSpec/BasicMaude.html

[24] TCH-Support: *Datentypen in C#.* http://tch-blog.com/?p=220

[25] Ullenboom, C.: *Imperative und funktionale Programmierung.* http://www.tutego.de/blog/javainsel/2013/05/imperative-und-funktionale-programmierung/

[26] Urban, W.: *Iteration und Rekursion.* http://www.hib-wien.at/leute/wurban/informatik/Rekursion.pdf