IGOR II – an Analytical Inductive Functional Programming System

Tool Demo

Martin Hofmann
Faculty Information Systems and Applied Computer Science
University of Bamberg
martin.hofmann@uni-bamberg.de

Abstract
The analytical inductive programming system IGOR II is an implemented prototype for constructing recursive functional programs from few non-recursive, possibly non-ground example equations describing a subset of the input/output (I/O) behaviour of a function. Starting from an initial, overly general program hypothesis, stepwise several refinement operators are applied which compute successor hypotheses. Organised as an uniformed-cost search, the hypothesis with the lowest costs is developed and replaced by its successors until the best does not contain any unbound variables.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.2 [Programming Techniques]: Automatic Programming; I.2.6 [Artificial Intelligence]: Automatic Programming; I.2.2 [Artificial Intelligence]: Learning

General Terms Inductive Programming, Functional Programming, Inductive Functional Programming

Keywords Inductive Programming, Tool Demo, Example, IGOR II

1. Introduction
Inductive Programming (IP) is the automatic construction of, probably recursive, programs from incomplete specifications, e.g. input/output (I/O) examples. As a special application of machine learning, IP systems create program hypotheses, that is, generalised, typically recursive, programs. Either, all correct programs are generated and tested against the specification, or incomplete programs are stepwise revised in a data-driven way guided by I/O examples. Systems following the further approach are called generate-and-test systems, the latter analytical systems. In contrast to classification learning, program hypotheses must cover all given examples correctly since for programs it is expected that a desired input/output relation holds for all possible inputs. Similar to machine learning each approach can be characterised by a preference and restriction bias [5].

In this tool demo paper, we introduce the analytical functional IP system IGOR II in its current HASKELL implementation. We refrain from a detailed formal description of the algorithm, but try to convey the idea behind IGOR II in an informal way as kind of comprehensive “hand simulation”. We introduce theoretical concepts where appropriate and/or refer to accordant papers.

2. The IGOR II-Algorithm – An Overview
IGOR II [1, 3, 4] is a prototype1 for constructing recursive functional programs from few non-recursive, possibly non-ground example equations describing a subset of the input/output (I/O) behaviour of a function to be implemented.

An IGOR II specification is correct module in a subset of HASKELL. It allows simple data type definitions and function bindings as shown in the following examples. It contains the data type definitions of all used types with its data constructors and the first $k$ non-recursive equations for the target function:

\[
\begin{align*}
\text{last} \quad &:: \ [a] \rightarrow a \\
\text{last} \quad &\left(a:[]\right) = a \\
\text{last} \quad &\left(a:b:[]\right) = b \\
\text{last} \quad &\left(a:b:c:[]\right) = c
\end{align*}
\]

Note, that the list type, as in HASKELL, is built in as a primitive so need not to be explicitly defined. A haskellish definition making the constructors apparent look as follows:

\[
\begin{align*}
data \ \text{List} \ a & = [] \ | \ a:(\text{List} \ a)
\end{align*}
\]

As in HASKELL, we will switch between the sweet $[a,b,c]$ and the sour $(a:b:c:[])$ representation where appropriate.

The output is a set of equations modelling the given I/Os:

\[
\begin{align*}
\text{last} \quad &\left(x:[]\right) = x \\
\text{last} \quad &\left(x_1:x_2:xs\right) = \text{last} \left(x_2:xs\right)
\end{align*}
\]

We refrain from formally defining the semantics of this HASKELL program as it should intuitionally be clear. Terms matching the left-hand side (lhs) of a rule, where a variable can subsume any subterm of the accordant type, can be replaced by the right-hand side (rhs) of this rule. This is continued until the term does not match a lhs anymore. Compared to other IP systems, IGOR II excels with a number of key features as

- termination by construction,
- handling arbitrary user-defined data types,
- utilisation of arbitrary background knowledge,
- automatic invention of auxiliary functions as sub programs,

1 The latest HASKELL version can be obtained as a CABAL package from http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'10, January 18–19, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-727-1/10/01…$10.00
learning complex calling relationships (tree-, nested recursion),
allowing for variables in the example equations,
simultaneous induction of mutually recursive target functions,
and the use of higher-order functions as program schemes.

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

**Initial Rule** The initial hypothesis contains one rule per target function. This rule is a least general generalisation (lgg) [6] of the example equations. The lgg for the previous **last**-examples is:

\[
\text{last } (x:xs) = y
\]

Without getting into theoretical details, it should be sufficient to know for now that constructor symbols or subterms occurring at the same position in all equations are kept, everything else substituted by variables. We say, that the rule \(\text{last } (x:xs) = y\) covers all previous examples, because the pattern on the lhs subsumes each lhs of the examples. Of course, this rule is not a functional program, because it contains an unbound variable on the rhs. To remedy this, the initial hypothesis is stepwise refined. IGOR II uses four transformation operators, which will now be briefly described, where section 3 will illustrate their application on a running example:

1. The I/O examples belonging to the open initial rule are partitioned into subsets and for each subset, a new initial rule (with a more specific pattern, or left-hand side (lhs), than the original rule) is computed.

2. If the open rhs has a constructor as root, i.e., does not consist of a single unbound variable, then all subterms containing unbound variables are treated as subproblems. A new auxiliary function is introduced for each such subterm.

3. The open right-hand side (rhs) is replaced by a (recursive) call to a defined function. The arguments of the call may be computed by new auxiliary functions. Hence, computing the arguments is considered as a new subproblem.

4. If it is possible to detect certain properties in the I/O examples a program schema as a higher-order function is introduced. Synthesising the argument function is a new induction problem.

**Splitting an open rule.** The first operator partitions the I/O examples belonging to a rule into subsets such that the patterns of the resulting initial rules are disjoint and more specific than the pattern of the original rule. Finding such a partition is done as follows: A position in the pattern \(p\) with a variable resulting from generalising the corresponding subterms in the subsumed example inputs is identified. This implies that at least two of the subsumed inputs have different constructor symbols at this position. Now all subsumed inputs are partitioned such that all of them with the same constructor at this position belong to the same subset. Together with the corresponding example outputs this yields a partition of the example equations whose inputs are subsumed by \(p\). Since more than one position may be selected, different partitions leading to different sets of new initial rules may result.

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

**Introducing a Program Schema** If the target function \(t\) obeys certain properties, we can re-express it using the **HASKE**l higher-order function **foldr** which implements structural recursion over lists as a catamorphism. Currently this is only possible for lists, but we plan to extend it to arbitrary inductive data types.

First, the target function must be of type \([α] → [β]\) and must be defined for the empty list \([\_]\). The output for the empty list is the default value for **foldr**. Further, there must be a function \(f\) such that \(t (x:xs) = f x (t xs)\). Finding such a function can be considered as a new induction problem, where the example equations for \(f\) can be abduced using this equality. Under certain conditions, such a **foldr** can be specialised to a **map**, a function applying an argument function to each element in a list, or a **filter**, a function selecting elements from a list which satisfy a provided condition.

### 3. Hand Simulation of the Algorithm

So let us demonstrate one run of our algorithm on a simple example. For now, we will only use the operators for partitioning (P), introduction of auxiliary functions (A), and call to background knowledge (C). We will consider the introduction of a higher order schema (HO) later (§3.6). Our example problem is the function **lasts** of type \([ [α] ] → [α]\), getting a list of lists and returning a list of all last elements.

\[
\begin{align*}
0. \text{lasts} & : [ [α] ] → [α] \\
1. \text{lasts} [ ] & = [ ] \\
2. \text{lasts} [[α]] & = [α] \\
3. \text{lasts} [[α],[β]] & = [β] \\
4. \text{lasts} [[α],[β],[γ]] & = [γ] \\
5. \text{lasts} [[β],[α]] & = [α,β] \\
6. \text{lasts} [[α],[β],[γ],[δ]] & = [δ,β] \\
7. \text{lasts} [[α],[β],[γ],[δ],[ε]] & = [ε,β,δ] \\
8. \text{lasts} [[α],[β],[γ],[δ],[ε],[ζ]] & = [ζ,β,δ,ε]
\end{align*}
\]

**Target Function**

Please note that we use some syntactic sugar for lists to keep the examples readable. The list \([α:b:c:d:ε]\), though, is nothing else than \((α:b:c:d:ε)\), where we use \((::)\) as an infix **cons** operator and switch between the two representations where appropriate.

As background knowledge we use the function **last** which we have already introduced above. Sure, IGOR II could solve the problem without additional help and with this function as additional knowledge, there is not much left for IGOR II but we want to keep the simulation within the scope of this paper.

\[
\begin{align*}
0. \text{last} & : [α] → a \\
1. \text{last} [α] & = a \\
2. \text{last} [α,b] & = b \\
3. \text{last} [α,b,c] & = c \\
4. \text{last} [α,b,c,d] & = d
\end{align*}
\]

**Background Knowledge**

The initial hypothesis is a single rule covering all examples of the target function, but with an unbound variable on the rhs.

\[
\text{last} x = y
\]

Initial Hypothesis \(H₀ : I\)
To keep track of the operator application we will indicate the development of a hypothesis \( H \) by applying operator \( O \) to the \( n \)th rule with \( H \circ \cdots \circ O \). Now we start to stepwise develop our initial hypothesis. In each iteration of the algorithm all available operators are applied to the currently best hypothesis.

### 3.1 Iteration 1

The initial hypothesis \( I \) is the only one in our search space at the moment, covering all example equation of \( \text{lasts} \).

#### Partitioning

We start with the partition operator. There is only the variable \( x \) on the lhs of the rule in \( I \). This rule is the lgg of rules \( \{ 1 \ldots 10 \} \) of the example equations of \( \text{lasts} \), which explains this variable, because rule 1 has the constructor \( [ ] \) on the position where rules 2 to 10 have the symbol \( . \). This induces a partition of all examples into the subsets \( \{ 1 \} \) and \( \{ 2 \ldots 10 \} \). Generalising both subsets, we get a new hypothesis with specialised patterns:

\[
\text{lasts} \quad [ ] = [ ]
\]
\[
\text{lasts} \quad ((x_0:x_1):x_2) = (y:y)
\]

\( H_1 : I \succ P \)

#### Auxiliary Introduction

The rule \( 1 \) in hypothesis \( I \) has a variable at the root position of the rhs, so the operator \( A \) is not applicable.

#### Function Call

In rule 1 of hypothesis \( I \) the rhs can not be replaced by a call to \( \text{lasts} \), because of different types. A recursive call to \( \text{lasts} \) would be conceivable, but since the argument must decrease in size to prevent non-termination this is not allowed here.

### 3.2 Iteration 2

Still, only one hypothesis, namely \( H_1 \), is in our hypotheses space and rule 2 is the sole open one.

#### Partitioning

The lhs of rule 2 contains three variables, so we can generate successor hypotheses partitioning w.r.t. to each of them.

The first variable \( x_0 \) does not induce a partition, because at this position all example equations contain a variable.

Partitioning w.r.t. the second, i.e., variable \( x_1 \), separates all rules which have a one-element list as first element as input, from those with more. The induced partitioning subsets are \( \{ 2 \ldots 5, 6, 9 \} \) and \( \{ 3, 4, 7, 8, 10 \} \) leading to a new hypothesis:

\[
\text{lasts} \quad [ ] = [ ]
\]
\[
\text{lasts} \quad ((x_0:x_1):x_2) = (x_0:x_2)
\]
\[
\text{lasts} \quad ((x_0:x_1):(x_2:x_3)) = ((x_0:x_1):x_2)
\]

\( H_2 : I \succ P \succ P_x \)

Partitioning w.r.t. the third variable i.e., variable \( x_2 \), separates all rules with two argument lists, from those with more. The induced partitioning subsets are \( \{ 5 \ldots 8 \} \) and \( \{ 9, 10 \} \) leading to a new hypothesis:

\[
\text{lasts} \quad [ ] = [ ]
\]
\[
\text{lasts} \quad [x_0:x_1] = [x_0]
\]
\[
\text{lasts} \quad ((x_0:x_1):(x_2:x_3):x_4) = ((x_0:x_1):x_2:x_3)
\]

\( H_3 : I \succ P \succ P_y \)

One can see that the partitions get more and more fragmented and finally will lead to an overfitting. However, IGOR II's bias is to prefer those hypotheses, which have the least number of partitions.

#### Auxiliary Introduction

The rule 2 of our current hypothesis \( H_2 \) has the infixed constructor symbol \( : \) at root position. So we can replace both subterms by calls to the auxiliary functions \( \text{fun}_1 \) and \( \text{fun}_2 \).

We do not have both of them, yet. To treat them as new induction problems, we need example equations for them. Consider \( \text{fun}_1 \) first. Using the input of our target function \( \text{fun}_1 \) has to compute the first element in the output list, leading to the following equations:

\[
\begin{align*}
\text{fun}_1 \quad [a] & = a \\
\text{fun}_1 \quad [c, a, b] & = c \\
\text{fun}_1 \quad [c, d, [e, f]] & = c \\
\text{fun}_1 \quad [c, d, [e, f], [g]] & = d \\
\text{fun}_1 \quad [[a, b], c] & = c \\
\end{align*}
\]

With these additional example equations for the new auxiliary functions we can develop our new hypothesis. The variables in the rhs of \( H_1 \)'s second rule have been replaced by calls to auxiliary functions. The initial rules for \( \text{fun}_1 \) and \( \text{fun}_2 \) have been included in the new hypothesis, too.

\[
\begin{align*}
\text{lasts} \quad [ ] = [ ]
\end{align*}
\]

\( H_4 : I \succ P \succ A \)

The \( \emptyset \) is not supported by IGOR II, but is a common syntax in HASKELL to bind a complex pattern to a simple variable. This keeps our code a little less messy.

#### Function Call

Because of type mismatch, a call to \( \text{lasts} \) is not allowed. Also not to \( \text{lasts} \), because we can not find both, a matching rhs and a smaller lhs for all equations covered by our rule 2.

### 3.3 Iteration 3

Now there are three hypotheses in the search space, but only \( H_4 \) has the least number of partitions. However, because both, the third and the forth rule, are open one is chosen arbitrarily. So we continue by developing rule 3.

#### Partitioning

Rule 3 of \( H_4 \) contains three variables. The first variable \( x_1 \) does not induce a partition, because at this position, all example equations contain a variable.

Partitioning w.r.t. to \( x_2 \) also induces two subsets of the example equation of \( \text{fun}_1 \), namely one where the first element is a singleton list \( \{ 1, 4, 5, 8 \} \) and the other where the first element is a list with at least two elements \( \{ 2, 3, 6, 7, 9 \} \).

\[
\begin{align*}
\text{lasts} \quad [ ] = [ ]
\end{align*}
\]

\( H_5 : I \succ P \succ A \succ P_z \)

Partitioning w.r.t. to \( x_3 \) also induces two subsets of the example equations of \( \text{fun}_1 \). One where all inputs are a singleton list \( \{ 1, 2, 3 \} \) and another where the input list has at least two elements \( \{ 4 \ldots 10 \} \).

\[
\begin{align*}
\text{lasts} \quad [ ] = [ ]
\end{align*}
\]

\( H_5 : I \succ P \succ A \succ P_z \)

Partitioning w.r.t. to \( x_3 \) induces two subsets of the examples of \( \text{fun}_1 \). One where all inputs are a singleton list \( \{ 1, 2, 3 \} \) and another where the input list has at least two elements \( \{ 4 \ldots 10 \} \).
we can close our current rule and make a new hypothesis

\( H \) of the covering rule

\( i \) = \( j \)

\( \text{Function Call} \) Considering the examples of function \( \text{fun} \), calls to \( \text{last} \) are due to type constraints not possible, but to \( \text{last} \). Matching the rhs of \( \text{fun} \) against the rhs of \( \text{last} \), \( \text{IGOR II} \) detects that it is possible to compute the output of \( \text{fun} \) by a call to \( \text{last} \). It also detects, that the argument for the call can be directly constructed from variables and constructors from the lhs of the covering rule 3. Thus, no auxiliary function is needed and the rhs can be replaced, which leads to a new hypothesis \( H \):

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

\( H_7 : I \geq 1 \) \( P \geq 2 \) \( A \geq 3 \) \( C_{\text{last}} \)

### 3.4 Iteration 4

The search space contains now five hypotheses \( H_2, H_3, H_5, H_6 \) and \( H_7 \), where all but \( H_7 \) have three partitions, which has only two. So the only rule in this hypothesis (rule 4) is developed.

### Partitioning

Because rule 4 in \( H \) has the same pattern as Rule 3 of \( H \), the patterns of the induced partitions are the same, only the rhs of the example equations covered by them are different, of course. The construction of resulting hypotheses \( H_5 \) and \( H_6 \) is straightforward and is left as an exercise to the interested reader.

### Auxiliary Introduction

This operator is again not applicable here.

### Function Call

Now it is only the function \( \text{lasts} \) to which a call is allowed and applicable and again, \( \text{IGOR II} \) can directly construct the argument for the call from variables and constructors from the lhs of the covering rule 4. So no auxiliary function is needed and we can close our current rule and make a new hypothesis \( H \):

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

\( H_{10} : I \geq 1 \) \( P \geq 2 \) \( A \geq 3 \) \( C_{\text{last}} \) \( C_{\text{lasts}} \)

### 3.5 Iteration 5

At the beginning of this iteration \( \text{IGOR II} \) finds the best hypothesis \( H_{10} \) closed. It still has only two partitions compared to the others with three. So \( H_{10} \) is returned as the final solution.

### 3.6 Alternative Iteration 1

However, we told you only half of the story, because we ignored the higher-order operator \( \text{HO} \) so far. Think back to the first iteration (§3.1), where only the operator \( P \) was applicable for the initial hypothesis \( I \). The \( \text{HO} \) operator will help us here even further.

Checking the conditions for \( \text{HO} \), we easily see that \( \text{lasts} \) is defined for the empty list \[]. Now we need to find a function \( \text{fun} \) such that \( \text{lasts} (x : xs) = \) \( x \) \( \text{lasts} \) \( xs \) for each example equation covered by the pattern on the lhs. We see that this is true for all example equations, namely \( 
\begin{align*}
1 & \rightarrow \text{fun} \ [x] \\
2 & \rightarrow \text{fun} \ [x_0] \ [x_1]
\end{align*}
\)

Thus, we can abdicate accordant example equations for our new auxiliary rule:

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

\( H_{2,alt} : I \geq 1 \) \( HO_{\text{map}} \geq 2 \) \( C_{\text{last}} \) \( C_{\text{lasts}} \)

### References


\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

Already now, we could rewrite the initial rule to \( \text{lasts} x = \text{foldr} \text{fun} \ [] \ x \), but we can simplify it even more. Note that the second argument of \( \text{fun} \) occurs unchanged in the output, so only the first argument is modified. The formatting of \( \text{fun} \) is supposed to illustrate this. The first argument of \( \text{fun} \) came from the first element in the argument list of \( \text{lasts} \), the second argument is the result of the recursive call with the rest list. Thus, \( \text{fun} \) modifies always the first element and inserts it at the front of the result list of the recursive call. This is exactly what \( \text{map} \) does as described above.

Therefore, it is admissible to always ignore the second argument for the auxiliary function \( \text{fun} \) in an alternative function \( \text{fun} \).

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

Instead of \( \text{foldr} \) we now use \( \text{map} \) to rewrite the initial rule and create a new hypothesis.

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

\( H_{1,alt} : I \geq 1 \) \( HO_{\text{map}} \geq 2 \) \( C_{\text{last}} \) \( C_{\text{lasts}} \)

### 3.7 Alternative Iteration 2

It is apparent from the example equations that \( \text{fun} \) is \( \text{lasts} \). So \( \text{IGOR II} \) will detect a call to the backbone knowledge as similarly already described in the third iteration (§3.3). We leave the details as an exercise and finish with the final solution, output after at the beginning of the third iteration.

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]

\( H_{2,alt} : I \geq 1 \) \( C_{\text{last}} \) \( C_{\text{lasts}} \)

\[
\text{fun} = \begin{cases} 
0 \text{ if } x = \text{last}(x_0) \\
\text{fun} \text{ otherwise}
\end{cases}
\]