Proceedings of AAIP 2011

4th International Workshop on Approaches and Applications of Inductive Programming

Emanuel Kitzelmann, Ute Schmidt (Eds.)

UNIVERSITY OF SOUTHERN DENMARK

Preface

Inductive programming is concerned with the automated construction of computer program code – typically including control structures like branching and recursion or loops – from incomplete specifications such as input/output examples. Inferred programs must be correct with respect to the provided examples in a generalizing sense: they should neither be equivalent to them, nor inconsistent. Applications in the focus of inductive programming are, among others, automated software development, algorithm design, end-user programming, cognitive modeling, and self-programming intelligent agents. Inductive programming is studied in different communities such as artificial intelligence, evolutionary computation, and programming languages and systems and has been tackled by different approaches like syntactic recurrence detection in sets of input/output terms, inductive reasoning, generate-and-test search in program spaces, and SAT and SMT solving.

The *Workshop on Approaches and Applications of Inductive Programming (AAIP)* series aims at bringing together researchers who are interested in inductive programming and to advance fruitful interaction between the different communities with respect to inductive programming approaches and algorithms, challenge problems, and potential applications. This year, AAIP took place in Odense, Denmark, July 19, in conjunction with the ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2011) and the International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011). Previous instances of AAIP took place in conjunction with the ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), the European Conference on Machine Learning (ECML 2007), and the International Conference on Machine Learning (ICML 2005).

This proceedings volume contains the five papers presented at the workshop. They cover machine learning approaches to inductive programming, genetic programming, and the combination of analytical and generate-and-test induction of recursive functional programs. The workshop was enriched by an invited talk by Ras Bodik from University of California Berkeley. His topic was *inductive program synthesis using SAT and SMT solving*, especially in the context of *Programming by Sketching*.

We thank the program committee members for a smooth review process and for providing meaningful reviews, the local organizers of the *Odense Summer on Logic and Programming* event for their organizational support, our invited speaker Ras Bodik for a revealing talk, and all authors of this volume for contributing to the AAIP workshop.

July 2011
Berkeley

Emanuel Kitzelmann
Ute Schmid
(AAIP Organizers)

Program Committee

Ricardo Aler	Universidad Carlos III, Spain
Pierre Flener	Uppsala University, Sweden
Lutz Hamel	University of Rhode Isalnd, USA
Jose Hernandez-Orallo	Universitat Politecnica de Valencia, Spain
Martin Hofmann	SAP Research, Germany
Johan Jeuring	Open Universiteit Nederland and Universiteit Utrecht, Netherlands
Susumu Katayama	University of Miyazaki, Japan
Emanuel Kitzelmann	International Computer Science Institute, Berkeley, USA
Pieter Koopman	Radboud University Nijmegen, Netherlands
Maria Ramirez	Universitat Politecnica de Valencia, Spain
Ute Schmid	University of Bamberg, Germany

Table of Contents

Applying distances between terms to both flat and hierarchical data

J.A. Bedoya-Puerta C. Ferri J. Hernández-Orallo
M.J. Ramírez-Quintana

DSIC, Universitat Politècnica de València
Camí de Vera s/n, 46022 València, Spain.
jorbepue@posgrado.upv.es, {cferri,jorallo,mramirez}@dsic.upv.es

Abstract. Terms are the basis for functional and logic programming representations. In turn, functional and logic programming can be used for knowledge representation in a variety of applications (knowledge-based systems, data mining, etc.). Distances between terms provide a very useful tool to compare terms and arrange the search space in many of these applications. However, distances between terms have special features which have precluded them from being used for other datatypes, such as hierarchical data or propositional data. In this paper, we explore the use of distances between terms in different scenarios: propositional data using the names of the attributes to construct the term tree (hierarchy), deriving the term tree by using attribute similarity and, finally, functional data representing hierarchies. In order to do this, we perform transformations from the original data representation to XML, thus allowing the use of term distances to directly handle objects of different degrees of 'hierarchisation', from flat data to fully hierarchical data.

Keywords: Distance functions, classification, term-based representation, hierarchical data, granular computing, XML data.

1 Introduction

Tree structures and functional terms have strong similarities. A term in functional (or logic) programming can be represented as an ordered tree. Trees can be used to represent information or knowledge (e.g. ontologies), and some very popular languages for information representation are based on trees or hierarchies, such as XML and related functional-alike structures [4]. So, inductive programming can be applied to this kind of information representation in a much more natural way than other paradigms which learn from examples with a flat structure (tabular data, text, etc.).

Although semantics are crucial to understand the role of a term or an atom wrt. a program, we can also analyse terms in an isolated, purely syntactical, way. In knowledge representation, terms and atoms may represent objects and its syntactical structure and content provides semantic information by itself.

Similarities (or dissimilarities) between objects are useful to understand how close they are. Distances (also called metrics) are measures of dissimilarity with some special properties such as symmetry and the triangle inequality, which make them more advantageous for many algorithms, because the search space can be reduced by triangularisation. Distance-based methods are a popular and powerful approach to inductive inference, since distances are a proper way to measure dissimilarity. The great advantage of these methods is that the same algorithm or technique can be applied to different sorts of data, as long as a similarity function has previously been defined over them [11].

There are distances for virtually any kind of object, including complex or highly structured ones, such as tuples, sets, lists, trees, graphs, images, sounds, web pages, ontologies, etc. One challenging case in machine learning, but more especially in the area of inductive programming, is the distance between first-order atoms and terms. Although atoms and terms can be used to represent many of the previous datatypes (and consequently, a distance between atoms/terms virtually becomes a distance for any complex/structured data), they are specially suited for term-based or tree-based representations. In this way, distances between atoms may not only be useful in the area of inductive logic programming (ILP) [12] (e.g. first-order clustering [3]) or inductive programming in general, but also in other areas where structured (hierarchical) information is involved such as learning from ontologies or XML documents. For instance, if an XML document represents a set of cars, or houses, or customers, we may be interested in obtaining the similarities between the objects, or to cluster them according to their distances.

In this paper, we define a *transformation procedure* to convert semi-structured data to a (functional) term-based representation in XML which is suitable for term distances. We handle different degrees of structure, from purely flat data (from which we derive a hierarchy in two different ways) to originally hierarchical data. The advantage of term distances is that they are able to consider context and, some of them, are able to consider repetitions. This may be important in some problems where the discrepancies of some arguments are more or less similar depending on whether they appear in other contexts and also the depth where discrepancy appears. We also need to address some issues about order in XML documents, since terms are always ordered. Once this transformation is done we can directly apply term-based distances.

We perform experiments with Nienhuys-Cheng's distance [13] and Estruch et al's distance [5]. We first apply our approach to two flat datasets from the UCI repository which have some implicit hierarchy that the transformation uses and the distances are able to exploit. Second, we use some techniques from granular computing to obtain a clustering (dendrogram) of attributes to construct a hierarchy which is again exploited by the term distances. We compare the results to an approach not using the hierarchical structure of these problems by using Euclidean distances. Third, we also work with an originally hierarchical problem, which is converted from a functional representation (in Lisp) to an XML representation. In all cases, experiments are performed with a distance-weighted

k-nearest neighbour algorithm, using several exponents for the attraction function (the distance weight). We see that in some cases these term distances can highly improve the results over a flat approach.

This paper is organised as follows. Section 2 introduces the notation and analyses some previous distances proposed in the literature for trees, semi-structured data and terms, and very especially the two distances we will use, Nienhuys-Cheng's distance [13] and Estruch et al's distance [5]. Section 3 introduces the transformation from semi-structured data into the common XML representation which is suitable for term distances. Section 4 includes experiments on several datasets showing the cases where these distances can work better than a flat Euclidean distance (in case the latter can be used). Section 5 concludes the paper and relates to future work.

2 Preliminaries and related work

2.1 Notation

Let $\mathcal{L}$ be a first order language defined over the signature $\Sigma = \langle \mathcal{C}, \mathcal{F}, \Pi \rangle$ where $\mathcal{C}$ is a set of constants, and $\mathcal{F}$ (respectively Π) is a family indexed on $\mathbb{N}$ (non negative integers) being $\mathcal{F}_n$ (Π_n) a set of $n-$adic function (predicate) symbols. Atoms and terms are constructed from the Σ as usual. An expression is either a term or an atom. The root symbol and the arity of an expression t is given by the functions $Root(t)$ and $Arity(t)$, respectively. Thus, letting $t = p(a, f(b))$, $Root(t) = p$ and $Arity(t) = 2$. By considering the usual representation of t as a labelled tree, the occurrences are finite sequences of positive numbers (separated by dots) representing an access path in t. We assume that every occurrence is always headed by a (implicit) special symbol λ, which denotes the empty occurrence. The set of all the occurrences of t is denoted by $O(t)$. In our case, $O(t) = \{\lambda, 1, 2, 2.1\}$. We use the (indexed) lowercase letters $o', o, o_1, o_2, \ldots$ to represent occurrences. The length of an occurrence o, $Length(o)$, is the number of items in o (λ excluded). For instance, $Length(2.1) = 2$, $Length(2) = 1$ and $Length(\lambda) = 0$. Additionally, if $o \in O(t)$ then $t|_o$ represents the subterm of t at the occurrence o. In our example, $t|_1 = a$, $t|_2 = f(b)$, $t|_{2.1} = b$. In any case, we always have that $t|_\lambda = t$. By $Pre(o)$, we denote the set of all prefix occurrences of o different from o. For instance, $Pre(2.1) = \{\lambda, 2\}$, $Pre(2) = \{\lambda\}$ and $Pre(\lambda) = \emptyset$. Two expressions s and t are compatible (denoted by the Boolean function $Compatible(s, t)$) iff $Root(s) = Root(t)$ and $Arity(s) = Arity(t)$. Otherwise, we say that s and t are incompatible ($\neg Compatible(s, t)$).

2.2 Distances over terms

Since in this paper we are not interested in dealing with non-ground terms, and we require a normalised distance which produces a single number in order to be able to compare term distances with Euclidean distances, we will only work with the following distances over atoms/terms: the Nienhuys-Cheng's distance

[13] and the Estruch et al.'s distance [5]. Apart from them, J. Ramon et al. [15] defined another distance between non-ground atoms that relies on the *least general generalisation* (*lgg*) operator [14] and that computes a pair of real values as the distance between two atoms. A comparison of the three distances in terms of different characteristics, namely context sensitivity, ability to take repeated differences, complexity of differences and variables into account, composability, use of weights and normalisation, can be found in [5]. As mentioned, J. Ramon et al's distance will not be used because it does not retrun a real number but a pair, and it cannot be used in many algorithms using distances.

In what follows we briefly review the Nienhuys-Cheng's distance and the Estruch et al.'s distance. The first one is a bounded distance which takes the context in that differences occur into account but disregarding the number of times that these differences take place or their syntactic complexity. On the other hand, the distance proposed by Estruch et al. is a bounded distance for (non-)ground terms/atoms which takes the context of the differences into account, but also their syntactic complexity and how many times these differences occur.

Definition 1. ***(Nienhuys-Cheng distance [13])*** *Given two ground expressions $s = s_0(s_1, \ldots, s_n)$ and $t = t_0(t_1, \ldots, t_n)$, the Nienhuys-Cheng distance between them, denoted by $d_N(s,t)$, is recursively defined as*

$$d_N(s,t) = \begin{cases} 0, & \textit{if } s = t \\ 1, & \textit{if } \neg Compatible(s,t) \\ \frac{1}{2n}\sum_{i=1}^{n} d(s_i,t_i), & \textit{otherwise} \end{cases}$$

For instance, if $s = p(a,b)$ and $t = p(c,d)$ then $d_N(s,t) = 1/4 \cdot (d(a,c) + d(b,d)) = 1/4(1+1) = 1/2$. Note that d_N takes the depth of the symbol occurrences into account in such a way that differences occurring close to the root symbols count more.

Besides the context, Estruch et al.'s distance also considers other factors related to the occurrences where the differences appear and their complexity. In order to do this, the authors defined the concept of the set of syntactic differences of expressions s and t as:

$$O^{\star}(s,t) = \{o \in O(s) \cap O(t) : \neg Compatible(s|_o, t|_o) \text{ and } Compatible(s|_{o'}, t|_{o'}), \forall o' \in Pre(o)\}$$

Then, the complexity of the syntactic differences between s and t is calculated on the number of symbols the subterms (in s and t) at the occurrences $o \in O^{\star}(s,t)$ have. For this purpose, a special function called $Size'$ is provided:

Definition 2. ***(Size of an expression)*** *Given an expression $t = t_0(t_1, \ldots, t_n)$, we define the function $Size'(t) = \frac{1}{4}Size(t)$ where,*

$$Size(t_0(t_1, \ldots, t_n)) = \begin{cases} 1, \; n = 0 \\ 1 + \frac{\sum_{i=1}^{n} Size(t_i)}{2(n+1)}, \; n > 0 \end{cases}$$

The context value of an occurrence o in an expression t, $C(o;t)$, is used to consider the relationship between $t|_o$ and t in the sense that, a high value of $C(o;t)$ corresponds to a deep position of $t|_o$ in t or the existence of superterms of $t|_o$ with a large number of arguments. This concept is formalised as follows:

Definition 3. (Context value of an occurrence) *Let t be an expression. Given an occurrence $o \in O(t)$, the context value of o in t, denoted by $C(o;t)$, is defined as*

$$C(o;t) = \begin{cases} 1, \ o = \lambda \\ 2^{Length(o)} \cdot \prod_{\forall o' \in Pre(o)} (Arity(t|_{o'}) + 1), \ \textit{otherwise} \end{cases}$$

It is proved in [5] that if $o \in O^\star(s,t)$ then $C(o;s) = C(o;t)$. So, in those cases, the context of an occurrence $o \in O^\star(s,t)$ is denoted as $C(o)$.

Repeated differences are handled through an equivalence relation ($\sim$) on the set $O^\star(s,t)$ defined as follows:

$$\forall o_i, o_j \in O^\star(s,t), \ o_i \sim o_j \Leftrightarrow s|_{o_i} = s|_{o_j} \text{ and } t|_{o_i} = t|_{o_j}$$

which produces a non-overlapping partition of $O^\star(s,t)$ into equivalence classes. Also, an order relation ($\leq$) in every equivalence class $O^\star_i(s,t)$ is defined as $\forall o_j, o_k \in O^\star_i(s,t), \ o_j \leq o_k \Leftrightarrow C(o_j) \leq C(o_k)$.

Additionally, the previous concepts are used to define another function which simply associates weights to occurrences in such a way that the greater $C(o)$, the lower the weight o is assigned, i.e., the less meaningful the syntactical difference referred by o is. Thus, given two expressions s and t, the weight function w is:

$$\forall o \in O^\star_i(s,t), \ w(o) = \frac{3f_i(o)+1}{4f_i(o)}$$

where $i = \pi(o)$, $\pi(o)$ is the index of the equivalence class o belongs to, and $f_i(o)$ is the position that o has according to $\leq$.

Finally, the distance by Estruch et al. is defined as:

Definition 4. ***(Estruch et al. distance [5])*** *Let s and t be two expressions, the distance between s and t is,*

$$d_E(s,t) = \sum_{o \in O^\star(s,t)} \frac{w(o)}{C(o)} \big(Size'(s|_o) + Size'(t|_o)\big)$$

For example, let $s = p(a,a)$ and $t = p(f(b), f(b))$ be two expressions. Then, $O^\star(s,t) = \{1,2\}$. Also, $C(1) = C(2) = 2 \cdot (2+1) = 6$.

The sizes of the subterms involved in the computation of the distance are:

$$Size'(a) = 1/4 \text{ and } Size'(f(b)) = 5/16$$

There is only one equivalence class $O^\star = O^\star_1(s,t)$. Assume that the occurrence 1 is ranked first, $w(1) = 1$ and $w(2) = 7/8$. Finally,

$$d_E(s,t) = \frac{1}{6}\big(\frac{1}{4} + \frac{5}{16}\big) + \frac{7}{48}\big(\frac{1}{4} + \frac{5}{16}\big)$$

2.3 Distances over trees and semi-structured data

The *edit distance* and the *tree alignment distance* are two well-known distances for comparing trees based on operations of deleting, inserting and relabeling nodes [2]. Given a cost function associated to each editing operation, the edit distance between two trees T_1 and T_2 is calculated as the cost of the optimal sequence of operations needed to transform T_1 into T_2, where a sequence is optimal if its cost in minimum. Analogously, the tree alignment distance between T_1 and T_2 is obtained by inserting nodes labeled by an empty label until both trees are isomorphic, then the resulting trees are superimposed giving a tree whose nodes are labeled by a pair of labels. Finally, the distance is given by the sum of the cost of each pair (using a certain cost function). Therefore, in both cases, the more complex the differences are, more symbols or nodes we need to edit or added, and, thus, the greater the computed distance is. In fact, many distances between trees that have been proposed only depend on this factor, since neither the presence of repeated differences nor their context have an effect on the value computed by the metric.

On the other hand, distances over semi-structured data (HTML, XML, and so on) have been mainly used for clustering documents and for detecting changes in XML documents. All these problems have been tackled by representing the documents as trees and, then, by applying tree distances. For instance, some approaches use the edit distance or some variant of it [17], [6], [18].

However, tree distances are not appropriate when each example is represented as a set, vector or hierarchy of features, since we need to align the features depending on their position and not merely by insert and delete operations. For instance, given the terms $t_1 = student(course('A'))$, $t_2 = student(course('B'))$ and $t_3 = teacher(course('A'))$, and considering the tree edit distance, t_2 and t_3 are at the same distance of term t_1 (only one relabeling operation is needed to transform t_2 or t_3 into t_1). However, t_1 and t_2 are "more similar" than t_1 and t_3 since they match up at the most external label (*student*). As we have shown in the previous section, for distances between atoms or terms, two subterms are just different when the topmost element of their tree representation is different. Also, the deeper the differences occur, the more similar the expressions are, and the lower their distance is. In this regard, we think that term distances may be more suitable, if the right transformation is applied to use them appropriately. Hence, in the rest of the paper we focus on them.

3 Transforming semi-structured data into a term-based representation using XML

Since we want to apply term distances to different kinds of data, from flat data to fully hierarchical data, we require a common language for representing all these types of data. Such a common language is XML.

Although we want to handle any degree of structure, we can distinguish two extreme situations:

– Flat data: Many datasets are given as a table of attribute-value pairs. However, a detailed inspection shows that some attributes are related to others and a hierarchy can be induced from them.
– Hierarchical data at source: Some other datasets consist of data with a hierarchical structure which is represented by a tree, a functional term (e.g. in Haskell or Lisp) or an “is-in” hiearchy.

Intermediate cases can be handled as well by using a mixture of the procedures.

3.1 Schema definition

XML documents and functional terms are not the same thing, so we need to make some decisions in order to use XML to represent functional terms, and also to adapt the previous types of data to a common representation. One of the key issues is how to handle depth, repetition and order, since some components in XML have order and some others not, and repetitions are allowed.

Given a hierarchy we have to be very careful when using term distances since some parts of the hierarchy might be empty, and we need to determine how an empty part is compared to a non-empty part. In other words, the structure may have different number of elements at the same level; therefore it is necessary to ensure that distance calculations are not affected by the absence of features or their order. This difficulty requires the creation of a general schema that allows each instance, with its own features, to be properly adjusted, without losing any element or content, and in a defined order. The schema we propose is an XML document that contains labels without content of all the elements that could exist for a given example.

```
...
<COLOUR>
  <INTERNAL>WHITISH</INTERNAL>
  <EXTERNAL>GREY</EXTERNAL>
</COLOUR>
...
```

```
...
<COLOUR>
  <INTERNAL>WHITISH</INTERNAL>
</COLOUR>
<PIGMENTATION>YES</PIGMENTATION>
...
```

Fig. 1. Hierarchies using differents elements.

For instance, Figure 1 shows two hierarchies with different elements. Thus, the element <EXTERNAL> does not exist in the first hierarchy (left), whereas the element <PIGMENTATION> exists for the second hierarchy (right), but not for the first one. It is therefore necessary to create a schema that contains an integrated structure for both cases. The schema created for this example is:

```
<COLOUR></INTERNAL/></EXTERNAL/></COLOUR></PIGMENTATION/>
```

This allows, not only to have a general structure, but also to ensure the format, so that each instance must follow the schema. However, it is necessary to consider that not all instances can be easily adapted to XML; e.g. elements that are leaves with content cannot be directly adapted when the schema requires a subtree. In this situation, the element is added regardless of the schema structure. However, this fact does not affect the distance calculations because they are considered as different features.

Even though there are differences between a representation based in terms and one represented in XML, it is possible to convert one representation into the other. In XML, every term is represented as an element composed by a label and a value. That way, it is possible to convert a term into an XML element adding a label and inserting the term as the value. It is also possible to convert an XML document into a term disregarding the label and representing the value as a term. For instance, in Figures 2 and 4 below, we see how an XML document can represent two different types of hierarchies.

3.2 Deriving hierarchical XML schemas from flat data

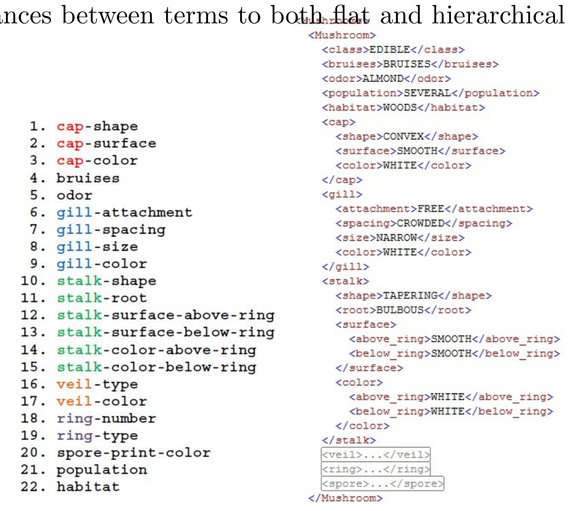
Flat data refers to attribute-value problems which are common in databases, machine learning, data mining, and other areas. In other words, data is presented in the form of a table of scalar values. In many cases, however, some structure can be inferred from these datasets, either because it was originally there and it still distills after the flattening process or because there are some features which can be grouped together according to some reason.

In fact, deriving a hierarchy from flat data is one of the problems that the area known as granular computing deals about [1]. In this work we consider three possible sources of structures from flat representations:

1. Value equality: Many datasets are given as flat data, but a detailed inspection shows that some attributes are related by the values they take. For instance, if two variables X_1 and X_2 can take the values $low, medium, high$, there is clearly a connection between them that can be exploited, especially through the use of equalities. For instance, we can define a condition or a rule using $X_1 = X_2$ which only makes sense if the datatypes are equal. This is the same as when variable repetitions are allowed in terms, like $f(a, X, X)$.
2. Name-induced hierarchies: In many cases we can find simple structures in the hierarchy of attributes, because of their names or their semantics. For instance, cap-shape, cap-surface and cap-color are attributes that contain 'cap' sub-features. In this way, sets of feature hierarchies can be created.
3. Attribute-similarity hierarchy: In other occasions, even if the names and values might be different, we can establish relations between the attributes which can be used to induce a structure. One approach is what is known as Watanabe-Kraskov variable agglomeration tree [16][9], which constructs a dendrogram (a hierarchical tree) using a similarity metric between attributes.

We will explore option 1 by the use (or not) of repetitions and, then, by applying the Estruch et al. distance and the Nienhuys-Cheng distance, respectively. We determine these relations in the Soybean dataset. The other two options will require specific transformations. Let us start with the name-induced hierarchy case.

Figure 2 shows an example of a hierarchy which is induced from the names of the original attributes. The dataset is 'mushroom', a flat dataset from the UCI machine learning repository [7], which has no structure originally (see Figure 2,



```
 1. cap-shape
 2. cap-surface
 3. cap-color
 4. bruises
 5. odor
 6. gill-attachment
 7. gill-spacing
 8. gill-size
 9. gill-color
10. stalk-shape
11. stalk-root
12. stalk-surface-above-ring
13. stalk-surface-below-ring
14. stalk-color-above-ring
15. stalk-color-below-ring
16. veil-type
17. veil-color
18. ring-number
19. ring-type
20. spore-print-color
21. population
22. habitat
```

```
<Mushroom>
  <class>EDIBLE</class>
  <bruises>BRUISES</bruises>
  <odor>ALMOND</odor>
  <population>SEVERAL</population>
  <habitat>WOODS</habitat>
  <cap>
    <shape>CONVEX</shape>
    <surface>SMOOTH</surface>
    <color>WHITE</color>
  </cap>
  <gill>
    <attachment>FREE</attachment>
    <spacing>CROWDED</spacing>
    <size>NARROW</size>
    <color>WHITE</color>
  </gill>
  <stalk>
    <shape>TAPERING</shape>
    <root>BULBOUS</root>
    <surface>
      <above_ring>SMOOTH</above_ring>
      <below_ring>SMOOTH</below_ring>
    </surface>
    <color>
      <above_ring>WHITE</above_ring>
      <below_ring>WHITE</below_ring>
    </color>
  </stalk>
  <veil>...</veil>
  <ring>...</ring>
  <spore>...</spore>
</Mushroom>
```

Fig. 2. Name-induced hierarchy for the mushroom dataset. Left: the original attributes. Right: the induced hierarchy using common names.

left). After a grouping of common name prefixes we get a hierarchy, which is finally represented as an XML document (Figure 2, right). This document can then be processed as a functional term, e.g.:

```
    Mushroom(EDIBLE,BRUISES,ALMOND,SEVERAL,WOODS,cap(CONVEX,SMOOTH,WHITE),
gill(FREE,CROWDED,NARROW,WHITE), stalk(TAPERING,BULBOUS,surface(SMOOTH,SMOOTH),
color(WHITE,WHITE)),veil(PARTIAL,WHITE),ring(ONE,PENDANT),spore(print(BROWN)))
```

A second approach, which does not rely on the names of the attributes, is based on the idea of finding the similarities among attributes (previously referred as option 3). In the case of numerical attributes, this is typically done through correlation measures. In the case of nominal attributes, other measures of association can be used, such as a chi-square test. In any case, once the similarity matrix is given, we can convert it into a dissimilarity matrix and construct a dendrogram, as mentioned above.

This process is illustrated in Figure 3, where we see (on the left) a dendrogram for the 22 attributes of the mushroom dataset, using the chi-square measure as (dis)similarity. From this dendrogram, instead of using the whole hierarchy, which would place some variables too deep in the hierarchy (and very low weight), we can just arrange them by using the longest segments into only four groups, as we see in Figure 3 (right). This process can be automatised [10].

From there, we place each variable in a group, leading to this term:

```
    Mushroom(g1(WHITE,WHITE, PENDAT, AMOND, WHITE), group2(g2(PARTIAL,WHITE,
SMOOTH, BROWN, SEVERAL), group3(g3(SMOOTH, WOODS, ONE, WHITE )), group4(
g4(TAPERING, FREE,CROWDED,NARROW, BRUISES, BULBOUS, CONVES, SMOOTH))))
```

3.3 Deriving hierarchical XML schemas from hierarchical data

Apparently, this situation seems more direct, but we need to determine whether order, repetitions and labels are relevant, in order to determine the features and

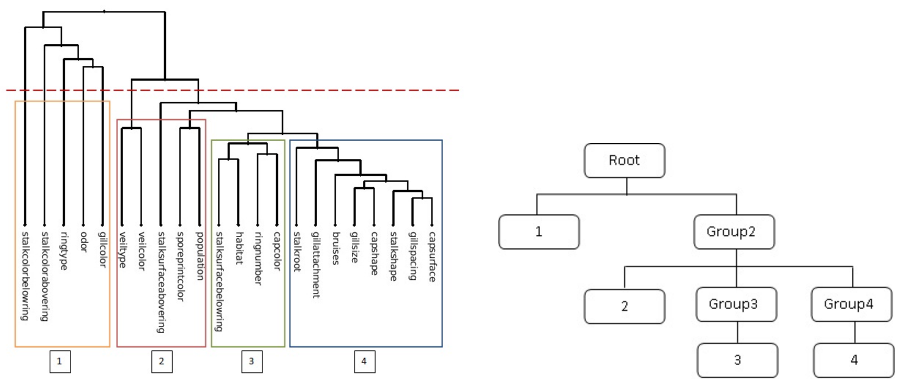


Fig. 3. Attribute-similarity hierarchy for the mushroom dataset. Left: the completed dendrogram. Right: the simplified hierarchy derived from it.

place them correctly in the XML document. A specific problem, as mentioned above, is how to treat empty parts in the hierarchy and how to compare them with non-empty parts. Our approach is based on a common schema for all the examples and the assumption that the based distance between an empty part and any non-empty part is 1 (and not given by the size of the non-empty part).

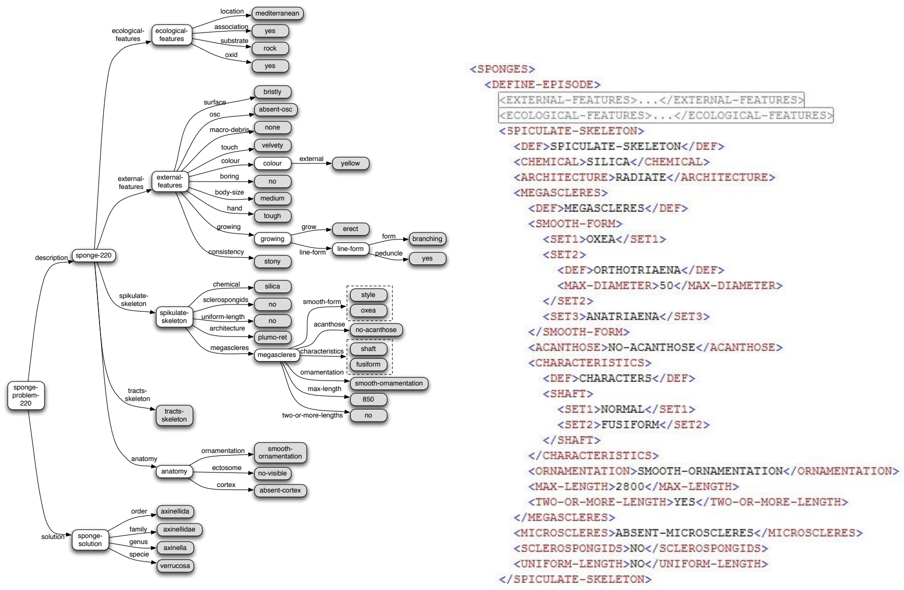
Figure 4 shows the ‘sponge’ dataset from the UCI repository [7], a complex dataset which enjoys a rich structure. Note that classical propositional methods are not applicable to this dataset. On the left of this figure we see the original hierarchy and on the right we see part of the resulting XML document (part of a single example). With this conversion, we can apply term distances.

4 Experiments

In this section we include some results using the distances introduced in section 2 with the several transformations seen in the previous section.

The simplest and most suitable algorithm for analysing the effect of the distances and transformations is the k-nearest neighbour (k-nn) algorithm, which just classifies an instance in the most common class of the k-nearest examples using the distance. The value of k in the following experiment is calculated as the square root of the number of examples, which is a common choice in k-nn. We use a weighted k-nn variant, using an attraction function which gives more or less weight to each of the k-most nearest examples, defined as $\frac{1}{d^i}$ where d is the distance and i is the attraction parameter. A non-weighted k-nn is just given when $i = 0$. The greater i is, the less important k is. In the experiments, we will use several values for i, varying from 0 to 3. We consider three distances: the Nienhuys-Cheng distance, the Estruch et al. distance and the Euclidean distance, which are denoted by d_N, d_E and d_U, respectively.

For the experimental evaluation, three datasets from the UCI repository [7] were used; two of them, Mushroom and Soybean, are flat and are used in several ways (flat and hierarchised by using two different methods); the third one, the



```
<SPONGES>
  <DEFINE-EPISODE>
    <EXTERNAL-FEATURES>...</EXTERNAL-FEATURES>
    <ECOLOGICAL-FEATURES>...</ECOLOGICAL-FEATURES>
    <SPICULATE-SKELETON>
      <DEF>SPICULATE-SKELETON</DEF>
      <CHEMICAL>SILICA</CHEMICAL>
      <ARCHITECTURE>RADIATE</ARCHITECTURE>
      <MEGASCLERES>
        <DEF>MEGASCLERES</DEF>
        <SMOOTH-FORM>
          <SET1>OXEA</SET1>
          <SET2>
            <DEF>ORTHOTRIAENA</DEF>
            <MAX-DIAMETER>50</MAX-DIAMETER>
          </SET2>
          <SET3>ANATRIAENA</SET3>
        </SMOOTH-FORM>
        <ACANTHOSE>NO-ACANTHOSE</ACANTHOSE>
        <CHARACTERISTICS>
          <DEF>CHARACTERS</DEF>
          <SHAFT>
            <SET1>NORMAL</SET1>
            <SET2>FUSIFORM</SET2>
          </SHAFT>
        </CHARACTERISTICS>
        <ORNAMENTATION>SMOOTH-ORNAMENTATION</ORNAMENTATION>
        <MAX-LENGTH>2800</MAX-LENGTH>
        <TWO-OR-MORE-LENGTH>YES</TWO-OR-MORE-LENGTH>
      </MEGASCLERES>
      <MICROSCLERES>ABSENT-MICROSCLERES</MICROSCLERES>
      <SCLEROSPONGIDS>NO</SCLEROSPONGIDS>
      <UNIFORM-LENGTH>NO</UNIFORM-LENGTH>
    </SPICULATE-SKELETON>
```

Fig. 4. Complex hierarchy for the sponge dataset. Left: the original hierarchy. Right: the hierarchy as an XML document.

Demospongiae (sponge) dataset, is a Lisp document which was transformed to XML by preserving the hierarchy and the values between attributes. For the Mushroom and Soybean datasets the tables include the means of a 10-fold cross validation experiment. In order to test the significance of these results, we perform a paired t-test between the methods (confidence 95%). If the difference of one method to another is significant, we include its acronym (d_N, d_E, d_U) in the cell. In the sponge dataset we used a 60% of training examples and a 40% for test examples.

4.1 Mushroom dataset

This dataset includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. First, we compare the three distances using a flat schema, i.e. without using any kind of hierarchy, and with different values of i. These results are shown in Table 1. Here, the differences between distances are not significant.

Table 2 shows the results using the hierarchy induced from the attribute names. With this setting, we can find some differences. The performance of Estruch et al's and Nienhuys-Cheng's distance is optimum, while Euclidean increases the accuracy when we increase the value of i.

Finally, using the other method of grouping by considering the similarity between attributes, based on the chi-square measure (ChiSquaredAttributeEval

	i=0 %	i=1 %	i=2 %	i=3 %
d_N	93.0	94.9	95.6	95.8
d_E	93.0	94.9	95.6	95.8
d_U	93.0	94.4	94.9	95.5

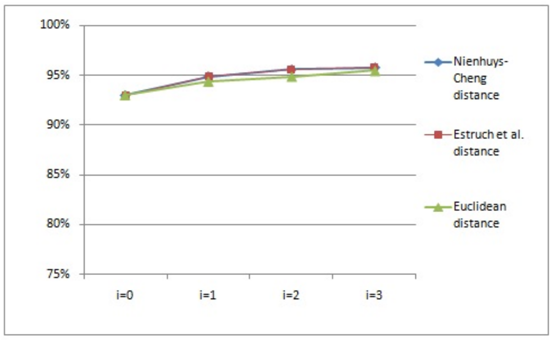


Table 1. Accuracies for mushroom without hierarchies.

	i=0 %	i=1 %	i=2 %	i=3 %
d_N	99.8	99.8	99.9	99.9
d_E	99.8	99.8	99.8	99.8
d_U	93.0	94.4	94.9	95.5

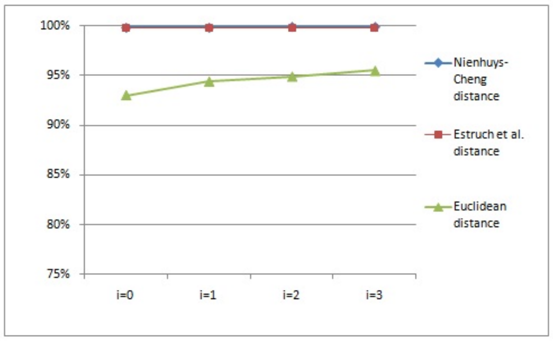


Table 2. Accuracies for mushroom with a hierarchy derived from the attribute names.

+ Ranker in Weka [8]) and the groups shown in Figure 3, we get the results shown in Table 3. Here, again, the distances between terms exploit this structure and get much better results than the Euclidean distance. Using similarity between attributes in the Euclidean distance could improve significantly its performance.

4.2 Soybean dataset

The soybean dataset is another ‘flat’ dataset and we do similarly as for the mushroom dataset. This dataset contains 307 instances and each instance has 35 attributes. Table 4 shows the results of three distances using the plain version of the dataset. In this case, the Euclidean distance obtains worse results than the two term distances. Table 5 shows the performance using a hierarchy which is derived from the attribute names. Here the behaviour of the three distances is similar and the differences are not statistically significant.

Finally, we used chi-square (ChiSquaredAttributeEval + Ranker in Weka [8]) to calculate the similarities, and grouping the variables from the dendrogram in

	i=0 %	i=1 %	i=2 %	i=3 %
d_N	99.5	99.8	100	100
d_E	99.5	100	100	100
d_U	93.0	94.4	94.9	95.5

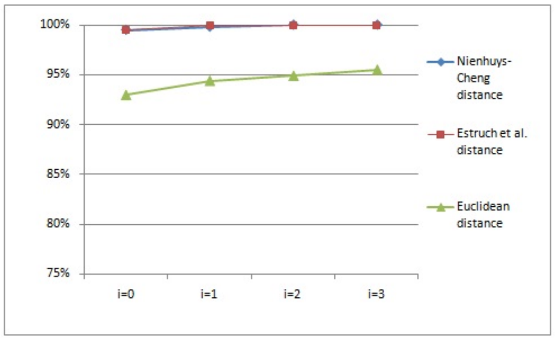


Table 3. Accuracies for mushroom with a hierarchy according to the similarity between attributes.

	i=0 %	i=1 %	i=2 %	i=3 %
d_N	75.3	91.3	93.9	95.8
d_E	75.3	91.3	93.9	95.8
d_U	75.3	87.1	91.3	92.6

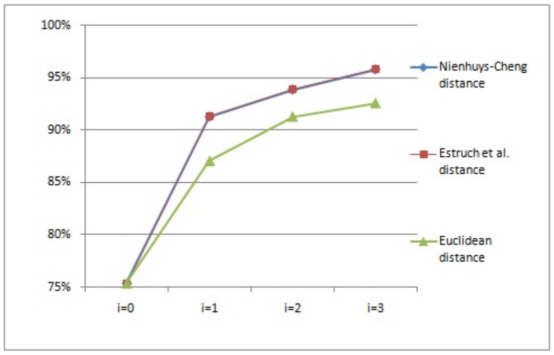


Table 4. Accuracies for soybean without hierarchies.

a similar way as we did with mushroom. With this process, we get the results shown in Table 6. In this setting, the atom-based distances obtain much better results than the Euclidean distance. Using similarity between attributes in the Euclidean distance could improve significantly its performance.

4.3 Demospongiae Dataset

Finally, we use the 503 examples of the Demospongiae dataset, which is originally hierarchical. Each instance is represented as a tree using terms in the language Lisp. Each tree has a depth between 5 and 8 levels and its number of leaves varies between 17 and 51 (see Figure 4).

From this dataset, a well formed XML structure that preserves the original structure was extracted, as discussed in the previous section. In order to do the transformation from Lisp, each line in Lisp was converted into one or several well-formed XML elements by assigning names and values to the elements for

	i=0 %	i=1 %	i=2 %	i=3 %
d_N	78.6	89.6	92.6	93.2
d_E	76.0	88.0	91.6	93.5
d_U	75.3	87.1	91.3	92.6

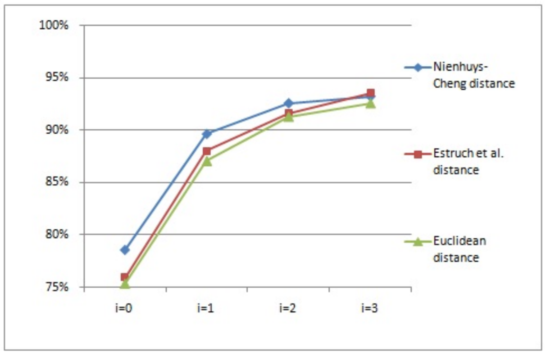


Table 5. Accuracies for soybean with a hierarchy derived from the attribute names.

	i=0 %	i=1 %	i=2 %	i=3 %
d_N	86.1 d_U	99.0 d_U	99.4 d_U	99.4 d_U
d_E	86.1 d_U	99.0 d_U	99.4 d_U	99.4 d_U
d_U	75.3 $d_N d_E$	87.1 $d_N d_E$	91.3 $d_N d_E$	92.6 $d_N d_E$

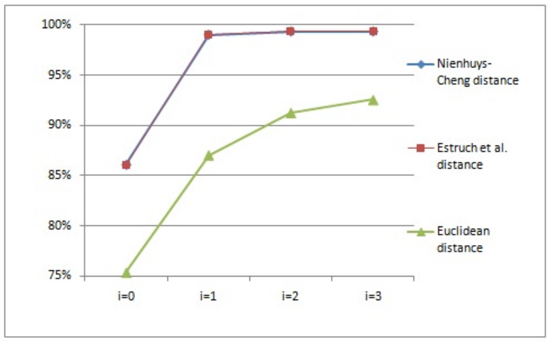


Table 6. Accuracy results of soybean with a hierarchy according to the similarity between attributes.

each feature described in Lisp in accordance to their hierarchy and order. Some specific processing was required:

- Each example in Lisp defined by label "define-episode" and an identifier "sponge :ID SPONGE-0" was transformed into an XML element: <DEFINE-EPISODE>, which in turn adds a child element: <SPONGE_ID>SPONGE-0</SPONGE_ID>.
- Each feature definition as "EXTERNAL-FEATURES DEFINE (EXTERNAL-FEATURES" was converted into a main element container of various elements in XML: <EXTERNAL-FEATURES>.
- Each simple feature such as "(BODY-SIZE SMALL)" was converted into an element <BODY-SIZE>SMALL</BODY-SIZE>.
- Additionally collections such as "(SET) GREY WHITISH", were converted to XML as follows <SET1>GREY</SET1> <SET2> WHITISH </SET2>.

Table 7 shows the classification results using the XML document. With this dataset we cannot apply directly Euclidean distance. We can see that (for values of i greater than 0) Estruch et al.'s distance shows a better accuracy than Nienhuys-Cheng's distance. This is the best example to illustrate the differences between these two atom-based distances because data is purely semi-structured, and we perceive the effect of repetition. In the end, this dataset is useful to take advantage of the intrinsic properties of atom-based distances.

	i=0 %	i=1 %	i=2 %	i=3 %
Nienhuys-Cheng distance	60.9	71.3	74.3	73.8
Distance between atoms	60.9	73.3	78.7	77.7

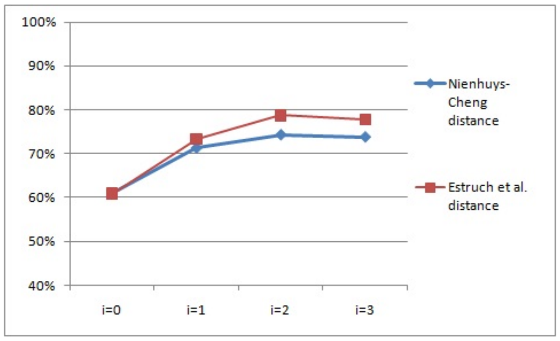


Table 7. Accuracies for the sponge dataset using its original hierarchy.

5 Conclusions

Tree distances are not generally appropriate for handling hierarchical datasets since they typically do not consider attribute names (the value for variable $X_3 = a$ can be matched with the same value for a different variable). In addition, they are based on the number of transformations to convert one tree into another, which is a meaningless concept in many hierarchy comparisons. We have seen that term distances, although initially defined in the area of inductive programming, can be used in a broader range of applications, provided we are able to find apropriate transformations for their term-based representation.

In this paper we have seen three transformations to adapt different degrees of structures and hierarchical data to be used with term distances. We have seen a method for constructing the hierarchy from the attribute names. This does not seem to provide good results. A second method uses the similarity between the attributes to construct a dendrogram from which a hierarchy is constructed. This second method improves the results of the flat dataset. A third, different transformation takes place when the original dataset has already a hierarchy. Here we can see the relevance of using repetitions or not. All these transformations are applied over XML schemas, so any dataset with any mixture or degree of flat and hierarchical information could be transformed and used with the term distances.

Summing up, we have seen a promising application of term distances to different types of datasets, which suggests that the use of term distances can be broader than it is now. In fact, as a future work, we plan to investigate the use of these distances for clustering and for other tasks. Nonetheless we believe that distances may have more potential applications in general inductive programming and also in other areas in programming language theory, such as debugging (as a measure of the magnitude of the error), termination (to find similar traces or similar rewriting terms), program analysis (to find similar parts in the code that could be generalised), and program transformation (to approximate the distance between two terms).

Acknowledgments

We thank the Spanish MICINN for projects TIN2010-21062-C02 and Consolider-Ingenio CSD2007-00022, and Generalitat Valenciana for project Prometeo/2008/051.

References

1. A. Bargiela and W. Pedrycz. *Granular computing: an introduction.* Springer, 2003.
2. Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computing Science*, 337:217–239, 2005.
3. H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proc. of the 15th International Conference on Machine Learning (ICML'98)*, pages 55–63. Morgan Kaufmann, 1998.
4. J. Cheney. Flux: functional updates for xml. In *Proc. 13th ACM SIGPLAN Intl. Conf. on Functional programming, ICFP*, pages 3–14. ACM, 2008.
5. V. Estruch, C. Ferri, J. Hernández-Orallo, and M. Ramírez-Quintana. An Integrated Distance for Atoms. *Functional and Logic Prog.*, pages 150–164, 2010.
6. F. De Francesca, G. Gordano, R. Ortale, and A. Tagarelli. Distance-based clustering of XML documents. In *1st Intl. Workshop on Mining Graphs, Trees and Sequences (MGTS-2003)*, pages 75–78. L. De Raedt and T. Washio, editors, 2003.
7. A. Frank and A. Asuncion. UCI machine learning repository, 2010.
8. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.
9. A. Kraskov, H. Stogbauer, R.G. Andrzejak, and P. Grassberger. Hierarchical clustering based on mutual information. *Arxiv preprint q-bio/0311039*, 2003.
10. Peter Langfelder, Bin Zhang, and Steve Horvath. Defining clusters from a hierarchical cluster tree. *Bioinformatics*, 24:719–720, March 2008.
11. T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.
12. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.
13. S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence.* Springer, 1997.
14. G. Plotkin. A note on inductive generalisation. *Machine Intelligence*, 5:153–163, 1970.
15. J. Ramon and M. Bruynooghe. A framework for defining distances between first-order logic objects. In *Proc. of the 8th Int. Conference on Inductive Logic Programming (ILP'98)*, pages 271–280. Springer, 1998.
16. S. Watanabe. *Knowing and guessing: A quantitative study of inference and information.* Wiley New York, 1969.
17. G. Xing, Z. Xia, and J. Guo. Clustering xml documents based on structural similarity. In *Advances in Databases: Concepts, Systems and Applications*, volume 4443 of *LNCS*, pages 905–911. Springer, 2007.
18. J. Y. Cai Y. Wang, D. J. Dewitt. X-diff: an effective change detection algorithm for xml documents. In *19th Intl. Conf.on Data Engineering*, pages 519–530, 2003.

Verified Stack-Based Genetic Programming via Dependent Types*

Larry Diehl

http://github.com/larrytheliquid/dtgp/tree/aaip11

Abstract. Genetic Programming (GP) can act as a powerful search tool for many kinds of Inductive Programming problems. Much research has been done exploring the effectiveness of various term representations, genetic operators, and techniques for intelligently navigating the search space by taking type information into account. This paper explores the less familiar concept of formally capturing the invariants typically assumed by GP implementations. Dependently Typed Programming (DTP) extends the type-level expressiveness normally available in functional programming languages to arbitrary propositions in intuitionistic logic. We use DTP to express and enforce *semantic* invariants relevant to GP at the level of types, with a special focus on type-safe crossover for strongly typed stack-based GP. Given the complexity involved in GP implementations and the potential for introducing logic and runtime errors, we hope to help researchers avoid erroneously attributing evolutionary explanations to GP run phenomena by using a verified implementation.

1 Introduction

The goal of this work is not to come up with a novel GP algorithm with respect to evolutionary performance, but rather give an example of a non-trivial but verified and simple-to-understand GP implementation. As GP algorithms and techniques increase in complexity and sophistication, it becomes more important to verify that the parts and the whole of the algorithm are doing what is expected. Towards this end we present the groundwork of basic verified GP, with special emphasis on correctness of the crossover operation.

While the earliest work in Genetic Programming used tree structures as candidate solutions to a problem, many alternative representations have been developed since (*e.g.*, linear, graph, grammar-based). Flat linear structures are conceptually simpler than nested trees and intimately familiar to functional programmers, yet still provide competitive evolutionary results compared to tree representations [9]. As such, we will concentrate on developing a stack-based genetic programming algorithm.

Researchers concerned with formal methods have produced many different theorem provers that could be used to prove GP correctness properties. However,

* Accepted for presentation at the 4th International Workshop on Approaches and Applications of Inductive Programming.

typical GP researchers are more familiar with programming languages than proof assistants. Dependently typed languages such as Agda [6] are a nice fit because they are expressive enough at the type level to enforce invariants present in GP, while retaining the look and feel of a programming language rather than a proof assistant.

After a general overview of stack languages and dependent types, the structure of the paper will follow a common classification scheme for GP:

- **parameters:** We will start with a non-dependently typed representation, and investigate how to use standard affair dependent typing to ensure the population size parameter is adhered to.
- **representation:** We will then modify our term representation to use precise dependent types, encoding arity information in the types of candidate programs.
- **evaluation function:** We will then introduce an evaluation function for evolved terms that is assured to terminate and not otherwise diverge, by taking advantage of the host language's totality requirement.
- **genetic operators:** We will then encode the property of transitivity into the types of functions related to crossover, ensuring that ill-typed programs never enter the population.
- **initialization procedure:** Finally, we will illustrate a basic procedure to initialize our population, taking care to only randomly select programs that match the type signature of the goal program.

1.1 Stack Languages

In stack-based languages such as Forth [3] there is no distinction made between "constants" and "procedures". Instead, each syntactic element is referred to as a "word". Every word can be modeled as a function which takes the previous stack state as a value and returns the subsequent, possibly altered, state. For example, consider a small language in the boolean domain, consisting of `true`, `not`, and `and`. A word such as `true` (that would typically be considered a constant) has no requirements on the input stack, and merely returns the input stack plus a boolean value of "true" pushed on top. On the other hand, `and` requires the input stack to have at least two elements, which it pops off and evaluates before pushing their logical conjunction back onto the stack to replace them.

For monotypic languages like our example, simple typing rules emerge which assign two natural numbers to each each word. The first represents the required input stack length (the precondition), while the second represents the output stack length (the postcondition). A sequence of such words forms a stack program, for which an aggregate input/output pair exists.

During genetic operations such as crossover, stack programs must be manipulated in some manner to produce offspring for the next generation. Tchnernev [8] showed how to use arity information related to the consumed/produced stack sizes to only perform crossover at points that will produce well-typed terms. Tchnernev [9] has documented many different approaches to do this, but for simplicity of presentation we will use 1-point crossover.

1.2 Dependent Types

Dependently typed languages allow arguments in type signatures to labeled (similar to value-level variable bindings) and used elsewhere in type signature to declare *dependencies* between types and values. This paper will use the dependently typed language Agda [6] for all of its examples. [1] Agda is a purely functional language like Haskell [2], but it is distinctively total (rather than partial) and has a more expressive type system (allowing the type-checker to enforce more properties).

At compile time, Agda programs must pass two checks to prove their totality. Termination checking is accomplished by checking for structurally decreasing recursive calls. Coverage checking is accomplished by requiring that every type-correct value of a function's arguments is accounted for in the function's definition. Consequently, Agda programs do not fail to terminate [2] or crash due to unexpected input.

Thanks to totality of the language, any "value level" function can also be used in type signatures to compute more precise typing requirements (without running into undecidability of type-checking issues).

2 Parameters

For purposes of pedagogy, we will first consider how to represent a population of terms/programs in a typical non-dependent functional programming style. Thereafter, we will extend the example to use dependent types. [3]

2.1 Population List

First, let's create a new type representing the possible words to be used for some evolutionary problem.

```
data Word : Set where
  true not and : Word
```

This simple example language is intended to operate on the boolean domain using well-known constants and functions. Of course, a stack program is not merely a single word, but a sequence of them that we would like to execute in order. The familiar cons-based list can serve as a container for several words, so let us type it out.

[1] It should be possible to translate examples to similar languages such as Epigram [4] or Idris [1].

[2] Agda programs can succeed to not terminate via coinductive definition and corecursion, if controlled non-termination is what we want.

[3] For a complete and proper tutorial on dependently typed programming in Agda, see [6]

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A

Term : Set
Term = List Word
```

Notice in particular the `A : Set` part of the list type. `Set` is the type of types in Agda, and `A` is a label that acts like a variable, but at the level of type signatures. In other words, we have created a polymorphic list type which is parameterized by the kind of data it can contain. `Term` is a specific instantiation of lists that can hold the `Word`s of our example language. Below are some examples of programs we can now represent.

```
notNot : Term
notNot = not :: not :: []

anotherTrue : Term
anotherTrue = not :: not :: true :: []

nand : Term
nand = not :: and :: []
```

GP requires us to work on not one but a collection of several terms, referred to as the **population**. Normally, this might be represented as a list of lists of terms.

```
Population : Set
Population = List Term
```

While the type above is certainly functional, it leaves room for error. This brings us to our first example of preserving some GP invariant with the help of dependent types. Namely, the population that GP acts upon is expected to be a certain size, and it should stay that size as GP progresses from one generation to the next.

2.2 Population Vector

In the dependently typed world, an easy and effective way to ensure that some invariant is held is to create a type that can only possibly construct values that satisfy said invariant ("correctness-by-construction"). In our case, we would like the population size parameter to be some natural number that we specify when configuring the run. This brings us to one of the canonical examples of a dependent type, the vector. We have already seen how the list type takes a parameter to achieve polymorphism. Vectors take an additional parameter representing their length.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

Population : ℕ → Set
Population n = Vec Term n
```

The empty vector has a constant length of `zero`. The length of a vector produced by "cons" is the `suc`cessor of whatever the length of the tail is. Given such an inductive definition of a type, the natural number index of any given vector can be nothing but its length. Just like our definition of `Term`, `Population` is just a specific instantiation of a more general type (`Vec`).

As an example, here is a small population of the three terms presented earlier.

```
pop : Population 3
pop = notNot :: anotherTrue :: nand :: []
```

Once again, note that the type requires a population of exactly three terms. If we were to supply any more or less, a type error would occur at compile time. We have effectively moved checking of certain *semantic* properties of our program to compile time, meaning much less can go wrong while the program is running. [4]

Now that we have seen how to construct a dependent type, let us see how a function operating on `Vec` can make use of its properties. During selection, GP will need to retrieve a candidate program from the population. An all-too-common error (taught even in introductory level programming courses) is indexing outside the bounds of a container structure. What means do we have to prevent this from occurring? Ideally, the type of the parameter used to lookup a member should have exactly as many values as the length of our vector. This way, a bijection would exist between the lookup index type and the vector positions.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)

lookup : {A : Set} {n : ℕ} → Fin n → Vec A n → A
lookup    zero (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

The type of finite sets `Fin` has exactly `n` possible values for any `Fin n`. In the `lookup` function the natural number index is shared between the finite set and vector parameters. The effect of this sharing is that every finite set argument has exactly as many possible constructions as the length of the vector argument,

[4] In fact, the only other causes for concern are logic errors due to bad encodings by the programmer. Typical runtime errors due to non-termination or lack of coverage are disallowed by the compiler.

statically preventing any “index out of bounds” errors from occurring. Since our `Population` is merely a specific kind of vector, we are able to use the safe `lookup` when defining a function for the selection process.

3 Representation

In the previous section we represented the terms in our population as unadorned lists of words. In order to perform type-safe crossover in a manner described by [8], the type of our terms will need to be more telling.

3.1 Typing Derivation

It should come as no surprise that when we implement a type-safe version of crossover, we will need to pay close attention to the types of the terms that we are manipulating. Just as `Vec` had an extra natural number parameter for its length, we desire a `Term` type with an extra parameter for the size of the consumed/input stack, and another for the size of the produced/output stack.

Before showing a generalized list-like `Term` type for arbitrary languages, we will take a look at a more traditional embedding of a typing relation into Agda.

```
data Term (inp : ℕ) : ℕ → Set where
  []   : Term inp inp
  true : {out : ℕ} → Term inp      out  → Term m (1 + out)
  not  : {out : ℕ} → Term inp (1 + out) → Term m (1 + out)
  and  : {out : ℕ} → Term inp (2 + out) → Term m (1 + out)
```

Recall that the first parameter is the consumed stack size and the second is the produced stack size. The empty term `[]` consumes some value `inp` and produces a stack of the same size, acting as an identity program. Note also that it has no premise, so it can be considered a type-theoretical axiom.

The other three constructors are parameterized by a previous `Term` value, representing the premise of each typing rule. This `Term` representation should be understood as follows: When considering `Term 2 1` as a type alone, `2` and `3` represent the input and output stack sizes respectfully. Within the context of a constructor with a `Term` premise, the “output” position of the premise represents that word’s *precondition* while the “output” position of the conclusion represent’s the word’s *postcondition*.

The `true` rule states that if we have some term which consumes some value `inp`, and produces another arbitrary value `out`, then the conclusion allows us to infer the existence of another term which has the same input and one additional output. In other words, `true` has a precondition that will always hold and a postcondition stating that the value in the precondition will be incremented by one.

In the `not` rule, the premise’s precondition requires that the previous output be more than just any arbitrary `out`. Instead, the previous output stack size must be at least one, but can be greater. Because the `out` parameter was given in braces, Agda treats this as an implicit argument that can be unified/inferred according to other types in context. In this way, `1 + out` can represent several values such as `1 + 0` or `1 + 7`. The conclusion of `not` allows us to infer the existence of the another term of output stack size `1 + out`. This fits with our informal mental model of `not` requiring at least one argument to pop off the stack, and pushing the logical negation back on.

Finally, `and` follows the same pattern, except it requires at least two values and produces just one, leaving the output stack size exactly one less than what it was previously.

As typing derivations, our previous list-based terms look like the following (note that we have overloaded the constructors of the `Word` and `Term` types).

```
notNot : Term 1 1
notNot = not (not [])

anotherTrue : Term 0 1
anotherTrue = not (not (true []))

nand : Term 2 1
nand = not (and [])

andAnd : Term 3 1
andAnd = and (and [])
```

Our terms now have the extra consumption/production values in their type. The `andAnd` term shows how the representation correctly composes the types of several terms. The first `and` requires two values and produces one, which satisfies one of the second `and`'s requirements, resulting in a final type of `Term 3 1`.

We can highlight that the input stack remains constant throughout subterms, with an exploded view of each of the subterms in `andAnd`.

```
a : Term 3 3
a = []

b : Term 3 2
b = and a

c : Term 3 1
c = and b
```

3.2 Syntactic Non-Uniqueness

To avoid confusion, we will point out that in our representation, multiple syntactically identical terms can have different types. Specifically, what can change is the original number of arguments on the stack that the bottommost empty constructor provides.

```
empty : Term 42 42
empty = []

nand’ : Term 6 5
nand’ = not (and [])

andAnd’ : Term 10 8
andAnd’ = and (and [])
```

Being able to represent multiple different types with a syntactically identical subterm is a property that we will later exploit when defining functions to safely split and recombine terms for crossover.

3.3 Derivation Abstraction

When writing functions over term types, it would be tedious to provide a case for every word in the language. Correspondingly, we will extract the common parts among the constructors of our language into a generic `Term`, which can be thought of as abstracting out each of the typing rules presented above.

The trick is to use module parameters for the type of `Word`s, as well as *functions* for the premise/precondition and conclusion/postcondition of each rule. The result is a generic list-like `Term` structure, and has the affect of making the library not tied to any particular language to evolve.

```
module DTGP (Word : Set) (pre post : Word → ℕ → ℕ) where

data Term (inp : ℕ) : ℕ → Set where
  []  : Term inp inp
  _::_ : {n : ℕ} (w : Word) → Term inp (pre w n) → Term inp (post w n)
```

```
pre  : Word → ℕ → ℕ
pre  true  n =     n
pre  not   n = 1 + n
pre  and   n = 2 + n

post : Word → ℕ → ℕ
post true  n = 1 + n
post not   n = 1 + n
post and   n = 1 + n

open import DTGP Word pre post
```

Just like a `List` or a `Vec`, our new `Term` now only has an empty case and a cons (_::_) case. Now we can rewrite our examples to look just like their `List` counterparts, except with the extra useful consumption/production natural numbers in their types.

```
notNot : Term 1 1
notNot = not :: not :: []

anotherTrue : Term 0 1
anotherTrue = not :: not :: true :: []

nand : Term 2 1
nand = not :: and :: []

andAnd : Term 3 1
andAnd = and :: and :: []
```

4 Evaluation Function

When comparing relative performance between evolved terms, one typically needs to evaluate them to determine fitness . We will proceed to write an evaluation function for the example language we have used so far. Rest soundly knowing that Agda will perform a termination and coverage check to prove the totality of functions. Notice that the example below has a case for every possible term and input vector, and uses the structurally smaller tail of the input term in recursive calls.

```
eval : {inp out : ℕ} → Term inp out → Vec Bool inp → Vec Bool out
eval [] is = is
eval (true :: xs) is = true :: eval xs is
eval (false :: xs) is = false :: eval xs is
eval (not :: xs) is with eval xs is
... | o :: os = ¬ o :: os
eval (and :: xs) is with eval xs is
... | o₂ :: o₁ :: os = (o₁ ∧ o₂) :: ns
eval (or :: xs) is with eval xs is
... | o₂ :: o₁ :: os = (o₁ ∨ o₂) :: os
```

In addition to the term to evaluate, `eval` takes a vector of booleans [5] whose length `inp` is equal to the number of inputs the term expects. The return type of the function is another vector of bools `out`, matching the evaluated term's output. Both of these properties are of course enforced statically, giving more assurance that our algorithm is doing what we expect.

4.1 Fitness Function

Once again, we use a module to accept a general scoring/fitness function as a parameter. Below is an example of a function that assigns a program (which

[5] Do not be confused by the true/false constructors of the `Bool` type and `Term` types. Agda can differentiate between overloaded constructor names, according to the type they have in context.

accepts two inputs and produces one output) a score equal to the number of provided examples for which it satisfies even parity.

```
module Evolution {inp out : ℕ} (score : Term inp out → ℕ) where
```

```
score : Term 2 1 → ℕ
score xs = count (λ is → head (eval xs is) == evenParity is)
  ((true :: true :: []) :: (true :: false :: []) ::
   (false :: true :: []) :: (false :: false :: []) :: [])

open Evolution score
```

5 Genetic Operators

When writing genetic operators, *e.g.* Tchnernev's [8] 1-point crossover, we need to take subsections of different terms and recombine them in a safe manner. Tchernev points out that we need to split parent terms at a point of equal output stacks to achieve safe recombination. This leads to a question: what is the criterion for a safe append of two arbitrary terms after they have been split in this manner?

5.1 Transitive Append

Terms may have different initial input stacks, and produce different outputs according to their contained words. A safe append of two terms illustrates the transitive property.

```
_++_ : {inp mid out : ℕ} → Term mid out → Term inp mid → Term inp out
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

bc : Term 2 1
bc = and :: []

ab : Term 3 2
ab = and :: []

ac : Term 3 1
ac = bc ++ ab
```

If an attempt is made to append two terms whose input and output requirements do not satisfy one another, a compile error will occur. Using a function with a such an informative type gives a high degree of confidence that we are doing the right thing, when used inside another function such as a crossover. As we shall soon see, the type of this function in fact gives us more than simple confidence.

5.2 Transitive Split

Now that we have a function to safely recombine terms in a transitive way, we need to come up with a compatible way to split a crossover parent. In DTP a *view* [5] is a general technique for using a specialized type to reveal structural information about another type. In our case, we want to view a term as another type representing the two subsections it was split into. The following is a derivative of the `TakeView` type in [7].

```
data Split {inp out : ℕ} (mid : ℕ) :
  Term inp out → Set where
  _++'_ : (xs : Term mid out) (ys : Term inp mid) → Split B (xs ++ ys)
```

The type above captures exactly how we would like split terms to be represented, such that they can be transitively recombined. The `mid` natural number index reveals the satisfied pre/post condition point a term was split at, and the term index is the value we are splitting. The constructor carries the two subterms which share `mid` in a way that the resulting type can recombine the two via `xs ++ ys`.

Given two parent terms split in such a way, crossover needs to produce two offspring that swap the subterms at the splits. Functions for both of these swaps can be straightforwardly defined.

```
swap₁ : {inp mid out : ℕ} {xs ys : Term inp out} →
  Split mid xs → Split mid ys → Term inp out
swap₁ (xs ++' ys) (as ++' bs) = xs ++ bs

swap₂ : {inp mid out : ℕ} {xs ys : Term inp out} →
  Split mid xs → Split mid ys → Term inp out
swap₂ (xs ++' ys) (as ++' bs) = as ++ ys
```

Dependent Pairs Given some term and a natural number, we would like to split the term at an indexed position represented by the number. This function will be the key to determining the split in the female parent of a crossover. `Split` is specific enough to tell us the shared `mid` between the two subterms. However, for the purposes of this function, we do not care what `mid` is (we would actually like for the function to determine the split point for us).

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

A non-dependent pair, or tuple, carries 2 values of arbitrary types. In the dependent version of pairs, the *value* in the first component is used to determine the *type* in the second component. One common DTP technique is to use a dependent pair to hide the index type of a return value when you don't know or care what it will be. For example, sometimes we would merely like to write down a vector value and have the compiler determine the unique possible length.

```
specifiedLength : Σ ℕ (λ n → Vec Bool n)
specifiedLength = 3 , true :: false :: true :: []

discoveredLength : Σ ℕ (λ n → Vec Bool n)
discoveredLength = _ , true :: false :: true :: []
```

Note the use of an anonymous function in the type. Remember that in DTP we can do anything at the type level that we can do at the value level, including the use of the intimately-known λ. With this dependent pair trick up our sleeves, we are prepared to define `split`.

```
split : {inp out : ℕ} (n : ℕ) (xs : Term inp out) →
  Σ ℕ (λ mid → Split mid xs)
split   zero  xs = _ , [] ++' xs
split (suc n) [] = _ , [] ++' []
split (suc n) (x :: xs) with split n xs
split (suc n) (x :: ._) | _ , xs ++' ys =
  _ , (x :: xs) ++' ys
```

Because we are returning a `Split` value, the split will always hold two subterms that can be transitively combined to produce the original. In this manner, splitting `andAnd` results in two `and :: []` values of type `Term 2 1` and `Term 3 2`.

5.3 Type-Preserving Crossover

With the previous types and functions defined, defining a crossover function that takes two parent terms of the same type and returns two child terms of the same type is not far away.

Split Female For the first step in 1-point crossover we need to split the first parent (referred to here as the "female") at some random[6] point. Thus, we need to know the length of the female, then choose a random number, bounded by that length.

```
length : {inp out : ℕ} → Term inp out → ℕ
length [] = 0
length (x :: xs) = suc (length xs)

splitFemale : {inp out : ℕ} (xs : Term inp out) → ℕ →
  Σ ℕ (λ mid → Split mid xs)
splitFemale xs rand with rand mod (suc (length xs))
... | i = split (toℕ i) xs
```

[6] To keep the example as simple as possible, here we pass the random number as a parameter to the function. The final implementation uses a standard `State` monad containing a random number seed for increased modularity and to avoid mistakenly reusing a random number.

Note that we use a `_mod_` function which returns a finite set representing the modulus of its two arguments. The definition of this function can be found in the supplementary source code, as it is not directly relevant to the explanation at hand.

Based upon the `mid` index at which the female was split, the male split can be determined by choosing a random member of all possible compatible splits.

```
splits : {inp out : ℕ} (n mid : ℕ) (xs : Term inp out) →
  Σ ℕ (λ n → Vec (Split mid xs) n)
splits zero mid xs with split zero xs
... | mid' , ys with mid =? mid'
... | yes p rewrite p = _ , ys :: []
... | no _ = _ , []
splits (suc n) mid xs with split (suc n) xs
... | mid' , ys with mid =? mid' | splits n mid xs
... | yes p | _ , yss rewrite p = _ , ys :: yss
... | no _ | _ , yss = _ , yss
```

Propositional Equality In the definition of `splits`, we simultaneously split at all possible positions within the male term, and filter out those possibilities that will not allow for a successful transitive recombination.

It is intuitive that the algorithm must compare the target `mid` of the original split to the `mid'` in the current split. Normally, a comparison of two terms is performed by passing them to a function that yields a boolean value, and handling the true and false cases differently. However, we need a richer version of the boolean type (the propositional equality type) whose values are associated with extra type-level information that can be used to make a `Split` value typecheck.

Consider the `yes p` case (analogous to a typical `true` case) within the `splits zero` case. We would like to return our freshly split `ys` value, but the type checker will not allow it. Why is this? If we look at the type signature of `splits`, it requires a `Split mid xs`, but `ys` is a `Split mid' xs`. Luckily the `=?` comparison function returned something more than just a boolean: it produced a constructive proof that both compared values were in fact the same. We pass the proof `p` (pattern matched as `yes p`) to Agda's `rewrite` keyword to convince the type checker that `ys : Split mid' xs` is acceptable because `mid ≡ mid'`.

What can we take away from all this? The primary point of interest is that the type checker requires formal constructive evidence in order to enforce invariants prescribed by the programmer. In practice, this evidence is easy to work with, as it is composed (as is everything else) of ordinary dependent types. The payoff is confidence; the burden of verifying that a program behaves as expected is lifted from the programmer's shoulders and onto the type checker's.

Split Male When we split the male parent, we choose a random member of the type-correct splits. However, this function returns a value of type `Maybe`, so that it may return `nothing` if there is no compatible split at all.

```
splitMale : {inp out : ℕ} (xs : Term inp out) →
  (mid rand : ℕ) → Maybe (Split mid xs)
splitMale xs mid rand
  with splits (length xs) mid xs
... | zero , [] = nothing
... | suc n , xss
  = just (lookup (rand mod suc n) xss)
```

Note that the proof complexity in the implementation of `splits` is isolated. Once we have a function definition that typechecks, we can freely use it without having to repeat any work.

Finally, we can write `crossover` to combine the female and male splits, and return both children using the `swaps` defined earlier.

```
crossover : {inp out : ℕ}
  (female male : Term inp out) (randF randM : ℕ) →
  Term inp out × Term inp out
crossover female male randF randM
  with splitFemale female randF
... | mid , xs with splitMale male mid randM
... | nothing = female , male
... | just ys = swap₁ xs ys , swap₂ xs ys
```

In the case where no valid male swap exists, we return the original two parents.

6 Initialization Procedure

At the onset of our GP run, we would like for our algorithm to operate on well-typed candidate programs. As such, the initialization function must be sure to only generate random type-correct programs with respect to our target program to evolve. By now, it should come as no surprise that we can (and will) enforce this requirement statically. A simple type-safe enumeration and filter strategy is adopted below.

6.1 Type-Safe Enumeration & Filter

First, we want to enumerate all terms up to some max length that conform to a given input stack size, `enum-inp`. Then, `filter-out` filters this result to include only those terms that match the desired output stack size, as well. The final list can be used as a pool to randomly select our population from.

```
enum-inp : (n inp : ℕ) → List Word → List (Σ ℕ λ out → Term inp out)
filter-out : {inp : ℕ} (out : ℕ) →
  List (Σ ℕ λ out → Term inp out) → List (Term inp out)
```

Dependent pairs are used once again, allowing us to return a list that is homogenous for `inp`, but heterogeneous for `out`. In order to implement this, we ask the user for a function that determines whether or not the precondition for a word that we want to extend a term with can be satisfied by the current output of said term.

```
module Initialization
(match : (w : Word) (out : ℕ) → Dec (Σ ℕ λ n → out ≡ pre w n))
where
```

Again, `Initialization` is another module, so the user is free to initialize the population by another means.

Decidable Relations `Dec` is a polymorphic type constructor whose values represent whether some proof of the type/proposition exists, or whether any such proof would lead to bottom ("bottom", or ⊥, is a type without constructors).

```
data Dec (P : Set) : Set where
  yes : ( p : P) → Dec P
  no  : (¬p : P → ⊥) → Dec P
```

`match` uses an existential proposition (dependent pair) inside `Dec`, and is *total* like all Agda functions. It effectively requires either a witness that the word's precondition satisfies the term's output, or a proof that no such satisfying value exists. This means that the implementor need not worry about the search for a suitable `n` ending too early, as can happen with `Maybe` (a type used commonly in this kind of situation).

```
match not zero = no ¬p where
  ¬p : Σ ℕ (λ n → 0 ≡ suc n) → ⊥
  ¬p (_ , ())
match not (suc n) = yes (n , refl)
```

The example above proves that when the output of a term is `0`, the precondition for `not` is unsatisfiable[7], and shows how to find a suitable `n` for any output greater than zero.

The definition of `enum-inp` plainly extends type-safe terms from the recursive call with the list of words argument (treating `Dec` similar to `Maybe`/partiality). `filter-out` is implemented even more straightforwardly, once again using `=?` to prove that the desired output is equal to what is returned.

[7] A pair of empty parentheses is Agda syntax used to indicate to the type checker that a value for this type is uninhabitable.

7 Conclusion

We have given an outline for a parameterized GP library whose operations are verified using dependent types. The same library can be used to evolve languages operating on domains besides booleans, such as the natural numbers, etc.

Dependent types can be used to enforce desired invariants by using informative data types and function type signatures. We have illustrated some basics for creating a verified stack-based GP implementation using type-safe 1-point crossover.

By building on a verified base, more complex GP algorithms can be created, and evolutionary data can be analyzed with much greater confidence that errors arising from implementation will not influence GP run behavior.

Hopefully, the examples presented herein can serve as a helpful template, to assist authors in encoding invariants for their particular flavors of GP within the context of dependently typed programming.

Finally, the techniques trivially extend to languages with multiple type stacks by parameterizing the main module over the domain type (e.g. $\mathbb{N} \times \mathbb{N}$), and providing a decision procedure for said type. Taking this technique further to evolve with arbitrary typing relations, rather than these Forth-like stacks, is currently under investiation.

References

1. E. C. Brady. Idris —: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM.
2. S. P. Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 2003.
3. M. G. Kelly and N. Spies. *FORTH: a text and reference.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
4. C. Mcbride. Epigram: Practical Programming with Dependent Types. pages 130–170. 2005.
5. C. Mcbride and J. Mckinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.
6. U. Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.
7. N. Oury and W. Swierstra. The power of pi. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.
8. E. Tchernev. Forth crossover is not a macromutation? In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 381–386, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
9. E. B. Tchernev. Stack-correct crossover methods in genetic programming. In E. Cantú-Paz, editor, *GECCO Late Breaking Papers*, pages 443–449. AAAI, 2002.

An Analytical Inductive Functional Programming System that Avoids Unintended Programs

Susumu Katayama

University of Miyazaki
1-1 W. Gakuenkibanadai, Miyazaki, Miyazaki 889-2192, Japan
skata@cs.miyazaki-u.ac.jp

Abstract. Inductive functional programming (IFP) is a research field extending from software science to artificial intelligence that deals with functional program synthesis based on generalization from ambiguous specifications, usually given as input-output example pairs. Currently, the approaches to IFP can be categorized into two general groups: the analytical approach that is based on analysis of the input-output example pairs, and the generate-and-test approach that is based on generation and testing of many candidate programs. This paper proposes a new analytical inductive functional programming system that generates, tests, and selects from many program candidates. For generating many candidate programs, the proposed system uses a new variant of IGOR II$_H$, the exemplary analytical inductive functional programming algorithm. This new system preserves the efficiency features of analytical approaches, while being robust to changes in the number of input-output examples while minimizing the possibility of generating unintended programs. In addition, this research can be considered a milestone in the fusion of both approaches in that it provides an analytical algorithm implemented in the same way as a generate-and-test algorithm and reveals the strengths and weaknesses of both approaches.

1 Introduction

Inductive functional programming (IFP) algorithms automatically generate functional programs from ambiguous specifications such as a set of input-output (I/O) example pairs or a loose condition to be satisfied by inputs and outputs. The term can include cases where no recursion is involved, as in genetic programming, but it usually involves generation of recursive functional programs.

Currently, two approaches to IFP are under active development. One is the analytical approach that performs pattern matching to the given I/O example pairs (e.g., [1] which was used for implementing the IGOR II system), and the other is the generate-and-test approach that generates many programs and selects those that satisfy the given condition (e.g., [2] which was used for implementing the MAGICHASKELLER system and [3] which was used for implementing the ADATE system). Analytical methods are efficient in general, but they have limitations on how to provide I/O relations as the specification, and, in general, the user has to provide many I/O examples, beginning with the simplest one(s) and progressively increasing their complexity. On the other hand, generate-and-test methods do not usually have limitations on the specification to be given (except, of course, that it must be written in a machine-executable form[1]), but tend to require more time compared to analytical methods.

This paper considers the improvements that can be made to the algorithm behind Igor II [1][4], which is the state-of-the-art analytical IFP system, to generate many program candidates by rewriting it using Spivey's monadic interface for combinatorial search [5]. The resulting algorithm has the following advantages:

– It can achieve the same properties as generate-and-test methods by generating all possible programs by using the fixed set of operators
– It could be combined with MagicHaskeller , which also uses Spivey's monadic interface for implementation.

The first point will be amplified by considering the example of synthesizing the *reverse* function, assuming it is not known how to implement it. Igor II requires the trace (or the set of all I/O pairs that appear during computation) of the biggest example as the set of examples, as in Table 1.[2] In this case, the first four lines are the computational traces of the last line and, hence, cannot be omitted. In general, it is tedious to write down all the required elements in order to generate a desired program, or a program with the intended behavior. Moreover, it is sometimes necessary to consider which examples are necessary in order to reduce the number for efficiency reasons. As a result, the user sometimes has to tune the set of examples until a desired program is obtained within a realistic time span.

On the other hand, MagicHaskeller , which is a type of generate-and-test system that uses systematic exhaustive search for generating programs, can generate a desired program from only one example of *reverse* $[1, 2, 3, 4, 5] \equiv [5, 4, 3, 2, 1]$. Obviously, this example has enough information to specify the intended function — the intention of the writer of this longer example is clear, while short examples such as *reverse* $[\,] = [\,]$ or *reverse* $[a] = [a]$ can be interpreted in many different ways.[3] MagicHaskeller often succeeds in generalization from only one example by the minimal program length criterion, or by selecting the shortest program that satisfies the given specification. Likewise, it can be expected that the same effect can apply even when using an analytical approach by analytically generating many candidate programs based on the given few examples and selecting those that satisfy one larger example, rather than generating a single candidate.

[1] Some readers may think that termination within a realistic time span is another limitation on the specification. Termination of the specification does not mean termination of the test process within a realistic time span, because the latter involves execution of machine-generated programs which may request arbitrary computation time. For this reason, time-out is almost indispensable for systems like MagicHaskeller , and in this case there is no such limitation on the specification.
[2] Throughout this paper, Haskell 's notation is used for expressions.
[3] Of course, the well-known *reverse* function is not the only function that satisfies the longer example. For example, there can be a case where we want to change the return value only for $[\,]$. This is not a problem, however, because no one would consider such a function by only giving the example of *reverse* $[1, 2, 3, 4, 5] \equiv [5, 4, 3, 2, 1]$, but most people would add the example for $[\,]$ in this case.

2 Preparation

This section introduces related IFP systems, IGOR II and MAGICHASKELLER .

2.1 Igor II

The algorithm behind IGOR II [1] synthesizes a recursive program that generalizes the given set of I/O examples by regarding them as term-rewriting rules through pattern matching. Early versions by Kitzelmann were written in MAUDE and interpreted, but recent implementations are in HASKELL , named IGOR II$_H$ [4], which is a simple port, and IGOR II$^+$ [6], which is an extension with support of catamorphism/paramorphism introduction. Such support is known to result in efficient algorithms, though this paper does not deal with those morphisms and, thus, is a counterpart of IGOR II and IGOR II$_H$.

These algorithms run in the following way:

1. Obtain the least general generalization of the set of the I/O examples by antiunification. This step extracts the common constructors and allows the uncommon terms to be represented as variables. Here, the same variable name is assigned to terms with the same example set. Variables that do not appear in the argument list represent unfinished terms.
2. Try the following operators[4] in order to complete the unfinished terms. Then, expressions with the least cost are kept, and others are abandoned. The cost function will be explained later in this section.
   Case partitioning operator introduces a case partitioning based on the constructor set of input examples, and tries this for each argument. Now, case bodies can include new unfinished terms. Each case can be finished by applying this algorithm recursively, supplying each field of the constructor application as additional inputs.
   Constructor introduction [5] operator introduces a constructor, if all output examples share the same one at the outermost position. Also introduce new functions to all fields, and supply the same set of arguments for that of the left hand side. Again, this part can be finished by applying this algorithm recursively, because it is possible to infer the I/O relation of the new function by reusing the same input example list and using each field of the constructor applications as output examples.
   Defined function call operator introduces either a function from the background knowledge (namely a predefined primitive function that works as a heuristic) if available, or a function already defined somewhere (causing a

[4] The term ‘operator’ is also used for ‘operator’ in ‘binary operator’. In order to avoid confusion, in the latter case, either its arity or HASKELL ’s operator name will always be mentioned, for example, ‘(+) operator’.

[5] This is usually called ‘introducing auxiliary functions’ (e.g., [6]), but in this paper it is called ‘constructor introduction’, because 1) it is the common constructor that is introduced specifically by this operator, 2) auxiliary functions are introduced even by other operators, and 3) the term ‘auxiliary function’ can be confused with the third operator.

Table 1. Example of synthesizing the *reverse* function using IGOR II$_H$: the input source text (taken from Version 0.7.1.3 of IGOR II$_H$ release) (left) and the resulting program (right)

```
reverse []           = []
reverse [a]          = [a]
reverse [a, b]       = [b, a]
reverse [a, b, c]    = [c, b, a]
reverse [a, b, c, d] = [d, c, b, a]
```

```
reverse []             = []
reverse (a0 : a1)      = fun1 (a0 : a1) : fun2 (a0 : a1)
fun1    [a0]           = a0
fun1    (_ : (a1 : a2)) = fun1 (a1 : a2)
fun11   (_ : (a1 : a2)) = a1 : a2
fun2    (a0 : a1)      = reverse (fun5 (a0 : a1))
fun5    [_]            = []
fun5    (a0 : (a1 : a2)) = a0 : fun9 (a0 : (a1 : a2))
fun9    (a0 : (a1 : a2)) = fun5 (fun11 (a0 : (a1 : a2)))
```

recursive call). These functions are called *defined functions* in both cases, and they are also represented as a set of I/O example pairs. Now, for each defined function f, the IGOR II algorithm tries to match the set of output examples that the unfinished term should return to that of f. Then, successful f's are adopted here.
Each argument of f is unknown, and thus a new function is introduced here. Again, it can finish this part by applying this algorithm recursively, because the I/O relation of the new function can be inferred by reusing the same input example list and using the input examples of the defined function as output examples.

Catamorphism introduction (optionally with IGOR II$^+$) introduces catamorphism. This can make some synthesis tractable, while it can slow down others. This operator is not yet included in the current implementation of the proposed algorithm.

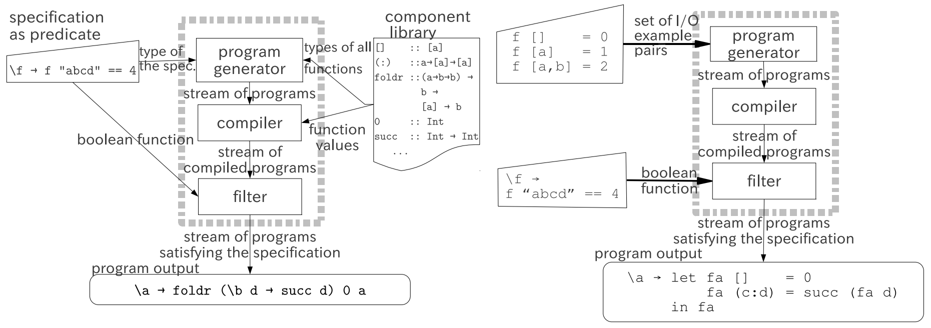
Table 1 shows an example of synthesizing the *reverse* function using IGOR II$_H$.

Limitations of Igor II The IGOR II algorithm has the following problems:

– IGOR II does not work correctly if we omit a line in the middle of the set of I/O examples; taking an example of *reverse*, if we omit the fourth line stating *reverse* $[a, b, c] = [c, b, a]$ from Table 1, it fails to synthesize correctly.
– There are many possible combinations while matching the target function to a defined function. Hence, an increase in the number of I/O examples easily slows down the synthesis.

Those problems incur a trade-off between the efficiency and the accuracy: in order to minimize the ambiguity a big example should be included in the example set; however, this means that all the smaller examples also have to be included, and as a result, the efficiency is sacrificed. This is problematic especially when examples increase in different dimensions. In fact, some functions such as multiplication cannot be synthesized by IGOR II$_H$ due to this trade-off. The proposed system solves this trade-off.

Cost and preference bias When searching in a broad space, in which priority order to try options is also an important factor in order to find answers in a realistic



```
\f → f "abcd" == 4
```

```
[]     :: [a]
(:)    ::a→[a]→[a]
foldr ::(a→b→b) →
        b →
        [a] → b
0      :: Int
succ   :: Int → Int
  ...
```

```
\a → foldr (\b d → succ d) 0 a
```

```
f []    = 0
f [a]   = 1
f [a,b] = 2
```

```
\f →
f “abcd” == 4
```

```
\a → let fa []    = 0
         fa (c:d) = succ (fa d)
     in fa
```

Fig. 1. The structure of MagicHaskeller (left) and the proposed system (right).

time span. Igor II defines a cost function that returns a tuple of the number of case distinctions, the number of open rules, etc., and the returned tuples are compared in lexicographical order. The search is implemented statefully by keeping track of the set of the best programs with the least cost.

Simply keeping track of the set of best programs is heap efficient, but that also means that second-best programs measured by the given cost function are abandoned, thus making it difficult to salvage a right program when the best programs are not actually those intended by the user.

2.2 MagicHaskeller

MagicHaskeller [7][8][9] is a generate-and-test method based on systematic search. One of its design policies goes “Programming using an automatic programming system must be easier than programming by one’s own brain”, and ease of use is its remarkable feature compared with other methods. Unlike other methods requiring users to write down many lines of programming task specification for each synthesis, users of MagicHaskeller only need to write down the specification of the desired function as a boolean function that takes the desired function as an argument. For example, the reverse function can be synthesized by only writing *printOne* $ $\lambda f \rightarrow f$ `"abcde"` $\equiv$ `"edcba"`. This is achieved by not using heuristics whose effectiveness is questionable and by enabling a general purpose primitive set (called a component library) that can be shared between different syntheses. On the other hand, because it searches exhaustively, its ability to synthesize big programs is hopeless. However, having heuristics and not doing an exhaustive search does not always mean that an algorithm can synthesize big programs, unless it is designed adequately and works well. According to benchmarks from the literature [10] and inductive-programming.org[6], at least it can be claimed that MagicHaskeller performs well compared with other methods.

Figure 1(left)depicts the structure of MagicHaskeller . Its heart is the program generator, which generates all the type-correct expressions that can be expressed by function application and λ abstraction using the functions in the given

[6] `http://www.inductive-programming.org/repository.html`

component library as a stream from the smallest and increasing the size. The generation is exhaustive, except that MagicHaskeller tries not to generate syntactically different but semantically equivalent expressions. The generation of expressions with the given type is equivalent to that of proofs for the given proposition under Curry-Howard isomorphism, and the MagicHaskeller algorithm [2] is essentially an extension of an automatic prover algorithm that can generate infinite number of proofs exhaustively.[9] MagicHaskeller adopts the breadth-first search for generating infinite number of proofs, and this is achieved by using a variant of Spivey's monad for breadth-first search [11]. All the generated expressions are compiled and tested by the given test function. By generating a stream of expressions progressively from the smallest and testing them, the most adequate generalization of the given specification that avoids overfitting comes first by the minimal program length criterion.

The component library corresponds to the set of axioms in a proof system under Curry-Howard isomorphism (e.g., [12]). It should consist of total functions including constructors and paramorphisms / catamorphisms, because permitting partial functions in the component library may make any type inhabited and causes search space bloat. As a result, MagicHaskeller with the default component library cannot generate partial functions without an inhabited type, such as $head :: [a] \to a$.

Early versions of MagicHaskeller try to detect and prune as many semantically equivalent expressions as possible by applying known optimization rules.[2] This involves guessing which in the component library are case functions, catamorphisms, or paramorphisms. Because such guessing does not work for user-defined types, this optimization was once removed, but now it is available by an option or by *init075* action.

3 Proposed Algorithm

As mentioned in Section 1, the proposed algorithm is an Igor II-variant that generates many program candidates, which is implemented using Spivey's monadic framework for combinatorial search [11][5]. For this reason, Spivey's framework is first reviewed and then the actual implementation is described.

When describing the algorithm, first the design policy and then the (somewhat simplified) algorithm are presented. The algorithm used for evaluation infers type while generating programs in order to narrow the search space, though this part is omitted in this paper.

3.1 Preparation: Monadic framework for combinatory search

Spivey [11][5] defined a very convenient interface to search strategies that simplifies the task of writing combinatory search algorithms. It can be defined as an extension of Haskell's *MonadPlus* class (which means that the structure of the search strategies is monadic and monoidal) with a new method *wrap*:

```
class MonadPlus m ⇒ Search m where
  wrap :: m a → m a
```

MonadPlus class has two methods inherited from its parent *Monad* class, namely $return :: forall\ a.\ a \to m\ a$ and $(\triangleright) :: forall\ a\ b.\ m\ a \to (a \to m\ b) \to m\ b$, and two monoidal methods, the unit *mzero* :: *forall a. m a* and the product *mplus* :: *forall a. m a* $\rightarrow$ *m a* $\rightarrow$ *m a*. As usual, *return* is used to wrap a normal value to make a search result, $\triangleright$ to combine two successive search operations, *mzero* to represent search failure, and *mplus* to try two alternatives (and *msum* to try a list of alternatives). In addition, *wrap* is used for degrading the priority of the current action.

Other monad-related functions from the standard library are also available. For example, the following is a code that enumerates all integers that are results of repeated addition, subtraction, and multiplication over 1, 2, and 3, ordered from those with small number of operators progressively.

```
nums ::    Search m ⇒ m Int
nums = fromList [1 . . 3] ‘mplus‘
           wrap (liftM3 ($) (fromList [(+), (−), (∗)]) nums nums)
fromList = msum. map return
```

This way, Spivey's monadic framework makes programming tasks of combinatorial search quite simple. One can enumerate expressions by enumerating function heads, enumerating their arguments by recursive calls, and applying them in the lifted way. Additionally, you may have to filter out those functions and subexpressions that do not satisfy the given constraints during enumeration.

In spite of the simplicity, this class covers useful search strategies like breadth-first search, depth-first search, and depth-bound search with/without iterative deepening. In addition, the design choice of using *wrap* method makes the algorithm applicable to best-first search without the idea of the depth in a search tree as long as the cost takes a natural number.

3.2 The design policy

The design policy is:

- First, candidates for the head (namely, the outermost function) explaining the examples are searched for, and then the spine (namely, the set of arguments) is formed for each of them by recursively synthesizing all the possible subexpressions to which each of the head candidates can be applied. This is in the same way as the *nums* example shown in Section 3.1. Also, MagicHaskeller is implemented in the same way.
- Case partitioning can be introduced by regarding case functions as the head; constructor introduction by regarding constructors as the head; and call for defined functions by regarding the functions themselves as the head.
- Antiunification has two effects: detecting common constructors and detecting subexpressions changing together; the former can be considered as a part of case partitioning and constructor introduction, and the latter can be implemented as a new operator **projection function introduction** which finalizes the synthesis of the current subexpression by finding an argument whose examples unify with the return value examples.
- Argument subexpressions are finished to form a spine by lazy recursive calls at once rather than explicitly representing unfinished expressions for now and then finishing them later by recursive calls as in Igor II_H . Since lazy evaluation

works, unbound variables can be identified with thunks, and it can be claimed that IGOR II$_H$ explicitly implements the lazy evaluation part of the proposed algorithm.
- All the search operators (except catamorphism introduction for now) are tried sooner or later, and this is implemented by taking their *msum*.
- For now, only natural costs are used. Cost n can be introduced by applying the *wrap* function (which is the function that lowers the priority by one depth in the Spivey's interface) n times. Case partitionings are assigned more cost than other operators because they cost in IGOR II . Constructor introduction and projection function introduction are assigned the least cost, because they correspond to antiunification, which supersedes other operators.

3.3 Outline of the algorithm

The simplified algorithm takes the tuple of the set of I/O pairs specifying the desired function and the sets of I/O pairs specifying defined functions, and it returns the prioritized set of generalized functions in the form of Spivey's search monad. If we call the function *synthesize*, it adds the current target I/O pairs to the set of defined functions, tries the four operators (which can be achieved by just adding the four computations with the *mplus* method), and calls the *synthesize* function recursively in order to synthesize the arguments by solving the induction problems from the I/O pairs that are presented by each of the operators.

3.4 Abstract definitions of the four operators

The proposed algorithm does not antiunify, and it has an additional operator, projection function introduction, instead. This section is devoted to providing with abstract definitions of these operators.

Projection function introduction is the simplest operator. Under the condition of

$$\forall j \in \{1...m\}.\ f\ x_{j1}\ ...\ x_{ji}\ ...\ x_{jn} = x_{ji}$$

it induces

$$\forall v_1 ... v_n.\ f\ v_1\ ...\ v_i\ ...\ v_n = v_i$$

This operator is tried for each argument $i \in \{1...n\}$.

Constructor introduction extracts a common constructor among the output examples of the I/O pairs of the given target function f. Under the condition of

$$\forall j \in \{1...m\}.\ f\ x_{j1}\ ...\ x_{jn} = C\ y_{j1}\ ...\ y_{jq}$$

it induces

$$\forall v_1 ... v_n.\ f\ v_1\ ...\ v_n = C\ (h_1\ v_1\ ...\ v_n)\ ...\ (h_q\ v_1\ ...\ v_n)$$

where

$$\forall i \in \{1...q\}. \forall j \in \{1...m\}. h_i\ x_{j1}\ ...\ x_{jn} = y_{ji}$$

and q is the arity of C. Further induction of $h_1 ... h_q$ from the newly introduced I/O pairs is required unless C is nullary.

In practice, C need not be a constructor but can be a function from a library. When this is permitted, synthesis from, e.g.,

$$\begin{aligned} & \mathit{sum}\ [\,] && = 0 \\ & \mathit{sum}\ [x] && = x \\ & \mathit{sum}\ [x, y] && = x + y \\ & \mathit{sum}\ [x, y, z] && = x + (y + z) \end{aligned}$$

is also possible.

Case partitioning focuses on an argument of the target function f, and puts together I/O pairs with such actual arguments that share the same constructor. Under the condition of

$$\begin{aligned} & \forall j \in \{1...q\}. \forall k \in \{1...p_j\}. \\ & \quad f\ x_{jk1}\ ...\ x_{jk(i-1)}\ (C_j\ z_{k1}\ ...\ z_{ko_j})\ x_{jk(i+1)}\ ...\ x_{jkn} = y_{jk} \end{aligned}$$

it induces

$$\begin{aligned} & \forall j \in \{1...q\}. \forall v_1 ... v_n . \forall u_1 ... u_{o_j} . \\ & \quad f\ v_1\ ...\ v_{i-1}\ (C_j\ u_1\ ...\ u_{o_j})\ v_{i+1}\ ...\ v_n = h_j\ v_1\ ...\ v_{i-1}\ v_{i+1}\ ...\ v_n\ u_1\ ...\ u_{o_j} \end{aligned}$$

where

$$\begin{aligned} & \forall j \in \{1...q\}. \forall k \in \{1...p_j\}. \\ & \quad h_j\ x_{jk1}\ ...\ x_{jk(i-1)}\ x_{jk(i+1)}\ ...\ x_{jkn}\ z_{k1}\ ...\ z_{ko_j} = y_{jk} \end{aligned}$$

Further inference of $h_1 ... h_q$ from the newly introduced I/O pairs is required. This operator is tried for each argument $i \in \{1..n\}$.

Note that this definition is slightly different from case partitioning of Igor II. In the proposed algorithm, the number of cases is equivalent to that of constructors that appear, while Igor II is more liberal about the number of cases and may put together I/O pairs with different constructors. Also, case partitioning of the proposed algorithm removes constructors, while that is done by antiunification of Igor II.

Defined function introduction matches the output examples of the target function f to those of a defined function g. Under the condition of

$$\begin{aligned} \forall j \in \{1...m\}.\ & f\ x_{j1}\ ...\ x_{jn} = \theta_j(w_{r_j}) \\ \forall i \in \{1...p\}.\ & g\ z_{i1}\ ...\ z_{iq} = w_i \end{aligned}$$

for existing substitutions $\theta_1 ... \theta_m$ and $r_1 ... r_m \in \{1...p\}$, it induces

$$\forall v_1 ... v_n.\ f\ v_1\ ...\ v_n = g\ (h_1\ v_1\ ...\ v_n)\ ...\ (h_q\ v_1\ ...\ v_n)$$

where

$$\forall k \in \{1...q\}. \forall j \in \{1...m\}.\ h_k\ x_{j1}\ ...\ x_{jn} = \theta_j(z_{r_j k})$$

Further inference of $h_1 ... h_q$ from the newly introduced I/O pairs is required. This operator is tried for each selection of defined function g and for each selection of $r_j|_{j \in \{1...m\}}$. Then, the loop checker checks if g is called with a "smaller" argument list in a well-formed sense than when g was first called.

3.5 Implementation of the four operators

The implementation of the four operators comes from interpretation of their abstract definitions.

All the applicable operators are tried, and if there are multiple possibilities within the operators, all of them are tried. This can be implemented by adding all the possibilities with *mplus*.

The applicability of each operator is decided by pattern matching. When the operators is defined in the format of "Under the condition of A it infers B (where C)" in Section 3.4, the target I/O examples for f is matched to A, and all the corresponding B's represent the induced programs.

Operators other than projection function introduction have the "where C" part, which defines the I/O examples for synthesizing the subexpressions recursively.

3.6 Efficient matching using a generalized trie

For each defined function and for each output example of the target function, the defined function introduction operator collects all the output examples of the defined function that the target output example matches to. The naive implementation of this process executes matching mn times for each defined function, where m denotes the number of I/O examples of the target function, and n denotes that of the defined function, and thus forms a bottleneck here. Our idea is to use the generalized trie [13] indexed by the output example expressions and to put all the I/O examples into the trie. Then, the n examples can be processed at once while descending the trie, by collecting values whose keys match the given expression. This is possible because the indexing of such generalized tries reflects the data structure of the index type, unlike hash tables.

Although it is difficult to be specific about the time complexity of the resulting algorithm, the algorithm reduces the computation time a great deal, and matching is not the bottleneck any longer.

4 Experimental Evaluation

This section presents the results of the evaluation of the proposed system empirically on its time efficiency and robustness to changes in the set of I/O examples. Here the term "the proposed system" means the system that executes the algorithm introduced in Section 3 and filters the resulting stream of programs with the user-supplied test function. (Figure 1(right)) The reader should note that users of the proposed system have to write the test function, as well as the I/O example pairs, while users of MagicHaskeller have only to write the test function.

4.1 Experiment conditions

Compared systems The proposed system was compared with the nolog release of Igor II_H Ver. 0.7.1.2, which is the latest nolog release (or release without logging overhead) at the time of writing, and with MagicHaskeller Ver. 0.8.5-1. Comparisons with other conventional inductive programming systems are omitted since comparisons between conventional systems including IGOR II_H and MAGICHASKELLER on the same programming tasks are already in the literature ([10] and inductive-programming.org[6]).

As for the search monad for the proposed system, based on preliminary experiments, Spivey's monad for breadth-first search [11] was selected over other alternatives that fit into Spivey [5]'s interface, such as depth-bound search and their recomputing variants, such as the *Recomp* monad [2].

MAGICHASKELLER was initialized with its *init075* action, which means that aggressive optimization without proof of exhaustiveness was enabled, like in its old stand-alone versions. By default, MAGICHASKELLER does not look into the contents of each component library function (or background knowledge function in the terminology of analytical synthesis) but only looks at their types. With *init075* action, however, it prunes the redundant search by guessing which are consumer functions such as case functions, catamorphisms, and paramorphisms, though some expressions with user-defined types may become impossible to synthesize due to language bias. This condition is fairer when compared with analytical approaches that know what case functions do.

Set of programming tasks Table 2 shows the test functions of the target functions used for filtering the generated programs. These test functions are higher-order predicates that the target functions should satisfy, and they were supplied to MAGICHASKELLER and the testing phase of the proposed system without modifications.

Their expected behaviors are not shown in this paper explicitly, mainly due to the page limit. They are explained in the benchmark site[6], though we believe that their test functions explain what the target functions are supposed to do and that usual human mind can generalize from the examples correctly with the hint of their names.

The left column of Table 2 shows the set of function names that were to be synthesized. They were selected by the following conditions:

– their I/O example pairs that are usable for synthesis are bundled in the IGOR II_H release, and
– they have already been compared with MAGICHASKELLER somewhere.

The second condition is about the adequacy of the task, and it was decided not to exclude those whose evaluation is temporarily postponed at the benchmark site[6]. Those programs that are too easy and require less than 0.5 second on all the systems were also excluded from the table. All of the other functions that were correctly answered by IGOR II_H within five minutes are included, provided that they satisfy the above conditions.

The first condition is included in order to fix the I/O example set by using those bundled as is. In analytical synthesis, the efficiency largely depends on the number of examples (except for the cases where the computation finishes instantly). For example, the set of I/O example pairs bundled in IGOR II_H for generating $(\equiv)$ compares two natural numbers between 0 and 2 in 9 ways — recursive programs could not be obtained if there were only 4 examples, while the computation would not be completed in a realistic time if there were 16 examples. Due to this problem,

Table 2. Test functions for target functions used to filter results from MagicHaskeller and the proposed system.

name	test function
addN	$addN\ 3\ [5,7,2] \equiv [8,10,5]$
allodd	$allodd\ [3,3] \wedge \neg\ (allodd\ [2,3]) \wedge allodd\ [1,3,5] \wedge \neg\ (allodd\ [3,7,5,1,2])$
andL	$\neg\ (andL\ [True, False]) \wedge andL\ [True, True]$ $\wedge\ andL\ [True, True, True] \wedge \neg\ (andL\ [False, True, True])$
concat	*concat* ["abc", "", "de", "fghi"] $\equiv$ "abcdefghi"
drop	*drop* 3 "abcde" $\equiv$ "de"
$(\equiv)$	$3 \equiv 3 \wedge \neg\ (4 \equiv 6) \wedge 0 \equiv 0 \wedge \neg\ (2 \equiv 0) \wedge \neg\ (0 \equiv 2) \wedge \neg\ (3 \equiv 5)$
evenpos	*evenpos* "abcdefg" $\equiv$ "bdf"
evens	$evens\ [4,6,9,2,3,8,8] \equiv [4,6,2,8,8]$
fib	$fib\ 0 \equiv 1 \wedge fib\ 1 \equiv 1 \wedge fib\ 3 \equiv 3 \wedge fib\ 5 \equiv 8 \wedge fib\ 7 \equiv 21$
head	*head* "abcde" $\equiv$ 'a'
init	*init* "foobar" $\equiv$ "fooba"
$(+\!\!+)$	"foo" $+\!\!+$ "bar" $\equiv$ "foobar"
last	*last* "abcde" $\equiv$ 'e'
lasts	*lasts* ["abcdef", "abc", "abcde"] $\equiv$ "fce"
lengths	*lengths* ["abcdef", "abc", "abcde"] $\equiv [6,3,5]$
multfst	*multfst* "abcdef" $\equiv$ "aaaaaa"
multlst	*multlst* "abcdef" $\equiv$ "ffffff"
negateAll	$negateAll\ [True, False, False, True] \equiv [False, True, True, False]$ $\wedge\ negateAll\ [False, True, False] \equiv [True, False, True]$
oddpos	*oddpos* "abcdef" $\equiv$ "ace" $\wedge$ *oddpos* "abc" $\equiv$ "ac"
reverse	*reverse* "abcde" $\equiv$ "edcba"
shiftl	*shiftl* "abcde" $\equiv$ "bcdea"
shiftr	*shiftr* "abcde" $\equiv$ "eabcd"
sum	$sum\ [7,3,8,5] \equiv 23$
swap	*swap* "abcde" $\equiv$ "badce"
switch	*switch* "abcde" $\equiv$ "ebcda"
take	*take* 3 "abcde" $\equiv$ "abc"
weave	*weave* "abc" "def" $\equiv$ "adbecf"

pragmatically it makes little sense to insist that an algorithm is quicker by some seconds if the example set is fine-tuned.

For this reason, the same set of I/O pairs as that included in Igor II_H -0.7.1.2 was used for analytical synthesis, namely, Igor II_H and the proposed system. That said, some I/O example sets bundled in Igor II_H -0.7.1.2 are obviously inadequate in that they seem not to supply enough computational traces. In Section 4.3, it will be shown what number of examples is enough and not too big for the corrected sets of examples.

No background knowledge functions were used by Igor II_H and the proposed system except the use of addition for the *fib* task.

Environment The experiments were conducted on one CPU core of the Intel® Xeon®CPU X3460 2.80 GHz. The source code was built with Glasgow Haskell Compiler Ver. 6.12.1 under the single processor setting.

Table 3. Benchmark results (left) and results for different number of I/O examples (right). "#exs." means the number of examples. Each number (except those below "#exs.") shows the execution time in seconds, rounded to the nearest integer. This is the time until the first program is obtained for MagicHaskeller and the proposed system. >300 represents that there was no answer in 5 minutes. Slashed-out numbers like $\cancel{0}$ mean that the result was wrong, that is, the behavior of the generated function to unspecified I/O pairs did not reflect the user's intension. ∞ means "impossible in theory" — this is only used for MagicHaskeller, when the requested function is a partial function without inhabited type and thus cannot be synthesized with the default primitive component set of MagicHaskeller.

	Igor II$_H$	MagH	proposed
addN	25	0	2
allodd	>300	4	>300
andL	0	0	1
concat	>300	3	>300
drop	>300	0	0
$(\equiv)$	3	22	0
evenpos	0	8	0
evens	$\cancel{0}$	93	>300
fib	>300	16	>300
head	0	∞	0
init	0	3	0
$(+\!\!+)$	3	0	0
last	0	∞	0
lasts	0	35	0
lengths	$\cancel{1}$	1	0
multfst	0	4	0
multlst	0	1	0
oddpos	0	8	0
reverse	0	0	0
shiftl	0	4	0
shiftr	0	42	0
sum	>300	0	>300
swap	0	>300	0
switch	0	>300	0
take	0	7	0
weave	>300	142	0

name	#exs.	Igor II$_H$	proposed
addN	3	$\cancel{0}$	> 300
	6	$\cancel{0}$	7
	9	$\cancel{0}$	6
	12	$\cancel{0}$	2
	15	$\cancel{0}$	0
	18	35	3
	21	> 300	> 300
allodd	6	$\cancel{0}$	> 300
	10	$\cancel{0}$	0
	15	> 300	26
	21	> 300	> 300
andL	1	$\cancel{0}$	> 300
	3	$\cancel{0}$	0
	7	0	0
	15	0	1
	31	> 300	> 300
concat	3	$\cancel{0}$	0
	6	$\cancel{0}$	0
	9	$\cancel{0}$	0
	13	> 300	> 300
drop	4	$\cancel{0}$	0
	6	$\cancel{0}$	0
	9	> 300	0
	12	> 300	0

4.2 Efficiency evaluation

The first experiment compares the efficiency of the proposed system with that of other systems using the same I/O examples as those bundled in the Igor II$_H$ release in order to make sure that the proposed system does not sacrifice the efficiency.

Table 3 (left)shows the benchmark results under the condition described in the previous section.

Comparisons between analytical systems The proposed system successfully avoids generating wrong functions by generating many programs and filtering them with a test condition. For all the cases where Igor II$_H$ generated the wrong result, it either returned a correct result or did not terminate. Since yielding a wrong result is just as misleading and no better than not yielding anything, at this point the proposed system is at least as good as Igor II$_H$.

In addition, the proposed system is as fast as or faster than Igor II$_H$ except when synthesizing *andL*, if the time required for human users to enter the test condition is ignored. The reason Igor II$_H$ is quicker than the proposed system on *andL* is because it specializes defined function introduction to *direct calls*, or calls with target function arguments.

On the other hand, the main reason the proposed system was faster than Igor II$_H$ is because a novel efficient algorithm for trying to match many expressions at once, which was presented in Section 3.6, was developed. This algorithm does not have a direct connection with Spivey's monad and could be applied to Igor II$_H$.

Comparison with MagicHaskeller When there are some case partitionings, MagicHaskeller tends to require more computation than analytical systems, which is why it cannot generate *swap* or *switch* in five minutes. Although both analytical systems and MagicHaskeller prioritize the search based on some cost functions, current versions of MagicHaskeller define the cost of a function as the number of function and constructor applications in the curried form, and thus having some functions with a bigger arity (like case functions) results in less priority. The cost function of MagicHaskeller may have room for tuning.

Also, MagicHaskeller with the default component library cannot generate partial functions without inhabited types such as $head :: forall\ a.\ [a] \to a$ and $last :: forall\ a.\ [a] \to a$.

On the other hand, since analytical systems cannot generate tail-recursive functions, they generate such functions in their linear recursive form. This sometimes results in unnecessarily complicated function definitions.

These facts could be suggested from benchmark results on conventional systems. However, by comparing Igor II$_H$ and MagicHaskeller with an analytical system implemented in the same way as MagicHaskeller , it has become even clearer whether each difference is due to that in the implementation or that in the paradigm. Now that it has been shown that there is an obvious difference in the strengths and weakness of both approaches, a fusion of both approaches will hopefully improve the overall performance.

4.3 Robustness to changes in I/O examples

The main purpose for adding a generate-and-test aspect to the analytical IFP is to obtain a system that works as expected for a variety of I/O example sets. In this section, the robustness of the proposed system to variation in the number of I/O example pairs is empirically evaluated in comparison with that of Igor II$_H$.

In this experiment, the raw sets of I/O examples from the Igor II$_H$ release were not used; rather, an edited version with enough computational traces was used, since several sets are tested for each target function. When n I/O example pairs are required, the first n examples of the longest set of I/O examples are used. For example, Table 4 shows the set of I/O examples used for synthesis of *addN*;

Table 4. Set of I/O examples of *addN* used for evaluating the robustness of the analytical systems.

$addN :: Int \to [Int] \to [Int]$			$addN\ 1\ [1]$	$= [2]$	
$addN\ 0\ [\,]$	$= [\,]$		$addN\ 1\ [2]$	$= [3]$	-- 12 examples
$addN\ 1\ [\,]$	$= [\,]$		$addN\ 1\ [0,0]$	$= [1,1]$	
$addN\ 2\ [\,]$	$= [\,]$	-- 3 examples	$addN\ 1\ [0,1]$	$= [1,2]$	
$addN\ 0\ [0]$	$= [0]$		$addN\ 1\ [1,0]$	$= [2,1]$	-- 15 examples
$addN\ 0\ [1]$	$= [1]$		$addN\ 2\ [0]$	$= [2]$	
$addN\ 0\ [2]$	$= [2]$	-- 6 examples	$addN\ 2\ [1]$	$= [3]$	
$addN\ 0\ [0,0]$	$= [0,0]$		$addN\ 2\ [2]$	$= [4]$	-- 18 examples
$addN\ 0\ [0,1]$	$= [0,1]$		$addN\ 2\ [0,0]$	$= [2,2]$	
$addN\ 0\ [1,0]$	$= [1,0]$	-- 9 examples	$addN\ 2\ [0,1]$	$= [2,3]$	
$addN\ 1\ [0]$	$= [1]$		$addN\ 2\ [1,0]$	$= [3,2]$	-- 21 examples

when synthesizing from six I/O example pairs the lines from $addN\ 0\ [\,] = [\,]$ to $addN\ 0\ [2] = [2]$ (and the line for the type signature) are used.

This experiment was only performed for the first five functions. Other conditions are the same as those in the previous section.

Table 3 (right)shows the results of the experiments. The results clearly show the merit of the proposed system, where, especially for *addN*, *andL*, *concat*, and *drop* the proposed system correctly generated a desired program from 3 or 6 examples and the test function, while IGOR II$_H$ is satisfied with the most simple program explaining the analyzed I/O example pairs and does not synthesize expected functions from a small set of examples. As can be seen from Table 4, the first 6 examples of *addN* simply return the second argument, and therefore IGOR II cannot do that even if a test process is added. On the other hand, the proposed system, residing between IGOR II$_H$ and MAGICHASKELLER , generates a desired program even in such a hard situation.

5 Conclusions and Future Work

An analytical IFP algorithm that can generate a stream of programs that generalize the given specification from the simplest to the least simple, instead of just generating the simplest program(s), was created.

By adding the generate-and-test feature to analytical synthesis, this algorithm solved the trade-off between the efficiency and the accuracy IGOR II had been suffering from. As a result, a desired program can be obtained without giving many I/O example pairs, and some functions that could not be synthesized analytically have become able to be synthesized.

In addition, by making the implementation of the new analytical IFP algorithm closer to that of MAGICHASKELLER , it has now become clear that both analytical and generate-and-test approaches have different strong points. This suggests that a complementary fusion of both approaches should be promising. As well, the threshold to fusing both approaches has been lowered. One option for fusing the software could be by adding a new operator that generates subexpressions by using MAGICHASKELLER and filters them by their I/O examples.

As for the efficiency, the new implementation is quicker than IGOR II$_H$ in most cases. However, as mentioned in Section 2.1, it should be noted that this result is not compared with the newest IGOR II$^+$. Further efficiency improvements using catamorphism or paramorphism introduction remain for future work.

Acknowledgements

Dr. Martin Hofmann kindly introduced the author to the implementation details of IGOR II$_H$ and answered the author's many questions. This work was supported by JSPS KAKENHI 21650032.

References

1. Kitzelmann, E.: Data-driven induction of recursive functions from input/output-examples. In: AAIP'07: Proceedings of the Workshop on Approaches and Applications of Inductive Programming. (2007) 15–26
2. Katayama, S.: Systematic search for lambda expressions. In: Sixth Symposium on Trends in Functional Programming. (2005) 195–205
3. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–81
4. Hofmann, M., Kitzelmann, E., Schmid, U.: Porting IgorII from maude to haskell. In Schmid, U., Kitzelmann, E., Plasmeijer, R., eds.: Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009. Volume 5812 of LNCS. (2010) 140–158
5. Spivey, J.M.: Algebras for combinatorial search. Journal of Functional Programming **19** (July 2009) 469–487
6. Hofmann, M., Kitzelmann, E.: I/O guided detection of list catamorphisms: towards problem specific use of program templates in ip. In: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation. PEPM '10 (2010) 93–100
7. Katayama, S.: Power of brute-force search in strongly-typed inductive functional programming automation. In: PRICAI 2004: Trends in Artificial Intelligence. Volume 3157 of LNAI., Springer-Verlag (August 2004) 75–84
8. Katayama, S.: Systematic search for lambda expressions. In: Trends in Functional Programming. Volume 6., Intellect (2007) 111–126
9. Katayama, S.: Recent improvements of MagicHaskeller. In Schmid, U., Kitzelmann, E., Plasmeijer, R., eds.: Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009. Volume 5812 of LNCS. (2010) 174–193
10. Hofmann, M., Kitzelmann, E., Schmid, U.: A unifying framework for analysis and evaluation of inductive programming systems. In: Proceedings of the Second Conference on Artificial General Intelligence. (2009)
11. Spivey, J.M.: Combinators for breadth-first search. Journal of Functional Programming **10**(4) (2000) 397–408
12. Barendregt, H.: Lambda calculi with types. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume 2., Oxford University Press (1992) 117–309
13. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10**(4) (2000) 327–351

Two New Operators for IGOR2 to Increase Synthesis Efficieny

Emanuel Kitzelmann

International Computer Science Institute
Berkeley, USA
emanuel@icsi.berkeley.edu

Abstract. Inductive program synthesis addresses the problem of automatically generating computer programs from incomplete specifications such as input/output examples. Potential applications range from automated software development to end-user programming to autonomous intelligent agents that learn from experience or observation. We present a recent version of the domain-independent algorithm Igor2 for the inductive synthesis of recursive functional programs, represented as rewriting rules. Igor2 combines classical *analytical* methods, that detect recursion by matching I/O examples, with *search* in program spaces as applied by recent generate-and-test methods; thereby widening the class of programs that are synthesizable in reasonable time. In particular, we present two recent improvements over an earlier Igor2 version which significantly increase the efficiency of the synthesis. Functions that were not inducible in several minutes are now induced in several seconds. It has already been shown that an earlier version of Igor2 outperforms other recent systems on several problems. In the empirical evaluation here, we show the significance of the improved synthesis operators by means of more complex problems, most of which were not tractable for Igor2 until now.

1 Introduction

Inductive program synthesis or *inductive programming (IP)* means the automated synthesis of programs where the problem specification, typically some examples of input/output behavior, is *incomplete*. IP has important application fields. For a recent example in end-user programming, see [4]. In [15] it is shown how IP can be used to model the cognitive capability of learning productive problem-solving knowledge in recursive domains. E.g., the general recursive strategy to solve *Towers of Hanoi* for arbitrary numbers of discs could be learned from solution traces for 1–3 discs.

IP and *supervised machine learning* have in common that a general concept or model is learned from I/O examples. However, unlike standard supervised learning [5], where a learned model maps objects to qualitative or quantitative values (*classification* and *regression*, resp.) and the models are non-recursive, in the case of IP, induced programs are typically recursive and not only is the (input) data structured (lists, trees etc.) but in general also the output. Further, we assume that examples are noise-free and require hypotheses to be consistent. Since recursion is a strong pattern, few data often suffices to learn the correct recursive function.

In this paper we are especially concerned with the synthesis of *recursive, functional* programs, represented as a special kind of term rewriting systems over first-order algebraic signatures. We describe a recent version of the Igor2 algorithm that leverages complementary strengths of two approaches to IP: *Analytical* methods detect recurrent patterns in I/O examples and generalize them to recursive functions [16, 9]. This is efficient but suffers from strong restrictions regarding the form of inducible programs and requires I/O examples that are complete up to some complexity (e.g., an input-list for each number of elements up to some maximum must be specified). *Generate-and-test* based systems [12, 7] generate lots of candidate programs and test them against given examples or evaluation functions. They overcome the strong restrictions of the analytical approach but suffer from unconstrained search in vast program spaces. Igor2 combines search in fairly unrestricted program spaces with analytical techniques to generate candidates and thereby widens the class of programs that are synthesizable in reasonable time. Igor2 is able to use background knowledge (BK) and automatically invents recursive subfunctions. It finds complex recursion schemes like that of *Ackermann* or the mutual-recursive definition of odd/even and is applicable in different domains.

In [8], a preliminary Igor2 version is described. We here review the general algorithm and then focus on the synthesis operators. In particular, we discuss some shortcomings and present extended versions that lead to a much more efficient synthesis and a wider class of tractable problems. The remainder of the paper is organized as follows: Section 2 introduces the Igor2 algorithm including its original synthesis operators. In Section 3 we discuss two synthesis operators more detailed and describe improved versions of them. In Section 4 we empirically evaluate the proposed new operators. Section 5 discusses some related work and in Section 6 we conclude.

2 The Igor2 Algorithm

We call functions that are to be synthesized *target functions*. Functions that are assumed to be implemented already and can be used are called *background functions*.

2.1 Representation Language

We briefly review basic term rewriting concepts as described, e.g., in [1].

Igor2 specifications of target and background functions as well as induced definitions of target functions are represented as *orthogonal* (see below) *constructor (term rewriting) systems (CSs)*. A CS is a set of (term rewrite) rules over a first-order algebraic signature (function symbols) and a set of variables, where the signature is partitioned into *defined functions* and *constructors* and where each rule has the form

$$f(p_1, \dots, p_n) \to t\,.$$

The symbol f is a defined function, the p_i are built from constructors and variables, and all variables in the right-hand side (RHS) t must also occur in the left-hand side (LHS) $f(p_1, \dots, p_n)$ ($\mathit{Var}(t) \subseteq \mathit{Var}(f(p_1, \dots, p_n))$). The argument constructor terms p_i are called *pattern*. We denote sequences of terms like $p_1, \dots, p_n$ by $\boldsymbol{p}$.

A CS is called *orthogonal* if its LHSs are linear, i.e., each variable occurs at most once in one and the same LHS, and pairwise non-unifying. Two terms are non-unifying if there is no substitution σ of variables by terms such that the terms become equal if σ is applied to both of them. Orthogonal CSs are a basic form of functional programs, excluding higher-order functions. Evaluation of an (input) term s is done by repeatedly matching subterms of it with LHSs of the CS—leading to substitutions σ of the pattern variables—and replacing the subterms by the respective RHSs with variables substituted according to σ. Orthogonality assures that if an evaluation terminates, i.e., reaches a *normal form*, then this normal form is unique. Hence orthogonal CSs denote (deterministic) functions. The (non-unifying) patterns of different LHSs for the same defined function act (i) as *conditions* to evaluate inputs of particular different forms differently and (ii) *decompose* a matching term into subterms. This concept is called *pattern matching* in declarative programming.

2.2 The Inductive Synthesis Problem

Specifications of target and background functions are orthogonal CSs with the restriction that the RHSs are built from constructors (and variables) only and hence are in normal form. *Ground* (no variables) specification rules denote I/O examples whereas specification rules containing variables represent sets of I/O examples given by all their ground instances. The inductive synthesis problem is defined as follows:

Definition 1 (Induction problem). *Let Φ and B be two specifications with disjoint sets of defined functions, $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, called* target functions *and* background functions*, respectively. Find a CS P with defined functions $\mathcal{D}_P$, such that*

1. *P is orthogonal,*
2. *P does not (re)define background functions,*
3. *for each $f(\boldsymbol{i}) \to o \in \Phi$, $P \cup B$ evaluates $f(\boldsymbol{i})$ to o.*

We use the *Rocket* problem [17], a simple benchmark problem in automated planning, as a running example. The problem is to transport a number of objects from earth to moon where the rocket can only move in one direction. The solution is to load all objects, fly to the moon and unload the objects. The assumption

Listing 1.1. Examples for the Rocket problem

```
rocket(nil, s)             → move(s)
rocket((o1 : nil), s)      → unload(o1, move(load(o1, s)))
rocket((o1 : o2 : nil), s) → unload(o1, unload(o2, move(load(o2, load(o1, s)))))
```

Listing 1.2. Strategy for rocket induced by Igor2

```
rocket(nil, s)       → move(s)
rocket((o : os), s)  → unload(o, rocket(os, load(o, s)))
```

now is that a planner (or an expert) already solved the problem for zero to two objects.[1] The problem instances and plans are then translated to example inputs and outputs for Igor2 (Listing 1.1). The objects are provided as a list (constructors nil, empty list, and an infix constructor _:_ to "cons" an object, 1st argument, to a list, 2nd argument). The variable s denotes a *state*, similar to situation calculus [10]. From the three examples, Igor2 induces the recursive strategy as shown in Listing 1.2.

2.3 General Search Strategy and Preference Bias

The induction of a solution CS is organized as a uniform cost search in spaces of orthogonal CSs, where the definition of CSs is relaxed in that $Var(r) \subseteq Var(l)$ need *not* be satisfied for all rules $l \to r$. We refer to CSs not satisfying this property, the respective rules, and their RHSs as *unfinished* and to CSs, rules, RHSs that satisfy it as *finished*. Either case might be meant in the following if we just say CS, rule etc. Unfinished CSs lead to non-unique normal forms and hence do not encode (deterministic) functions. Purpose of the synthesis operators is to transform an unfinished CS into a finished one. Igor2's refinement operators, described in the next section, assure that *all* constructed candidate CSs P satisfy Def. 1. Hence each *finished* CS is a solution.

The cost of a candidate CS is defined as the number of disjoint patterns in it, hence CSs that correctly compute the examples based on fewer case distinctions have lower cost and are preferred. The initial candidate CS consists of one single initial rule (see below) for each target function.

2.4 Initial Candidate Rules and CSs

As initial hypothesis for a set of specification rules, Igor2 takes their *least general generalization (LGG)* [13]. That basically means that if all rules have the same symbol at a particular position, it is kept for that position in the LGG,

[1] The instance for zero objects is a bit artificial. We chose it to keep the example as simple as possible, but it may be skipped and replaced by the three objects instance.

and if the symbols differ, a variable is introduced, where it is assured that the same variable is introduced at different positions, if the corresponding subterms in the example rules are the same at both positions. The LGG for the `rocket` examples (Listing 1.1) is

```
rocket(os, s) → s' .
```

The variable `os` results from the different constructors `nil` and `_:_` at the same position in the example rules. The variable `s'` results from the different symbols `move` and `unload`. This initial rule is *unfinished* due to the variable `s'`, which does not occur in the LHS. Hence it will be refined.

2.5 Synthesis Operators

In general, if a candidate CS P is chosen during the search, one of its unfinished rules r is selected to be refined. One refinement consists of a set s of successor rules. IGOR2 applies three operators independently to an unfinished rule r to compute refinements: (i) It splits r into sets of at least two new initial rules with disjoint patterns that are more specific than the pattern of r; (ii) it considers unfinished *sub*terms of the RHS of r as new subproblems; (iii) it replaces the RHS of r by (recursive) function calls. Assume a CS P is chosen and let $r \in P$ be the selected unfinished rule. Applying the refinement operators results in a finite (possibly empty) set $\{s_1, \ldots, s_n\}$ of successor-rule sets s_i. For each s_i a successor candidate CS P_i is generated by $P_i = (P \setminus \{r\}) \cup s_i$.

Rule Splitting. Consider the example rules in Listing 1.1 and the corresponding initial rule, `rocket(os, s) → s'`. The pattern variable `os` results from the different constructors `nil` (1st example) and `_:_` (2nd, 3rd example). We call a position that denotes a *variable* in the LHS of an initial rule and *constructors* in the LHSs of the corresponding example rules a *pivot position*. Now the splitting operator χ_{split} partitions the examples according to the different constructors at the pivot position. In our example, the first example rule goes into one subset and the remaining two into a second one. The refinement of the unfinished initial rule then consists of a set of new initial rules, one for each subset of the generated partition. In our example:

```
rocket( nil , s)       → move(s)
rocket((o : os), s)  → unload(o, s')
```

Since the new initial rules always contain the different constructors at the pivot position in their LHSs, they are non-unifying. Since χ_{split} increases the number of disjoint patterns in a CS, it increases the cost of a candidate.

If more than one pivot position exists, this probably leads to different partitions and different refinements, all of which are returned by χ_{split}. In Section 3.1 we discuss this operator more detailed and propose a variant that (i) makes larger refinement steps and (ii) is deterministic. This leads to a more efficient synthesis process as empirically shown in Section 4.

Dealing with Unfinished Subterms Separately. Consider the initial rule `rocket((o:os), s) → unload(o, s')` for example rules 2, 3 that resulted from splitting the original initial rule and which is unfinished due to variable `s'`. Since `s'` occurs as a proper subterm in the RHS, it can be dealt with as a subproblem. Therefore, the subproblem operator χ_{sub} replaces `s'` by a call to a new subfunction `sub`,

```
rocket((o : os), s) → unload(o, sub((o : os), s)),
```

and takes as examples for it the appropriate subterms of the RHSs of the corresponding `rocket` examples:

```
sub((o1 : nil), s)        → move(load(o1, s))
sub((o1 : o2 : nil), s)   → unload(o2, move(load(o2, load(o1, s))))
```

The refinement step is finished by computing an initial rule for `sub` and adding it to the `rocket` hypothesis:

```
rocket(nil, s)        → move(s)
rocket((o : os), s)   → unload(o, sub((o : os), s))
sub((o : os), s)      → s'
```

The operator χ_{sub} is deterministic and only defined if the RHS of the selected rule is rooted by a constructor. Even though χ_{sub} adds new rules to a candidate, it does not increase its cost because the added rules always only contain patterns already present in the candidate CS.

Introducing (Recursive) Function Calls. Introducing (recursive) function calls with appropriate arguments is the most complex operation. For an unfinished rule $f(\boldsymbol{p}) \to t$, the function call operator χ_{call} produces refinements of the form $f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$, where f' is some already defined function (a target-, background- or previously introduced subfunction; probably $f = f'$) and the g_i are new defined functions to be induced subsequently. The idea behind the g_i as arguments (instead of just constructor terms over pattern variables) is that the arguments in the call of f' possibly need to be computed by another, possibly new and/or recursive, subfunction. As an example consider the *Quicksort* algorithm where the arguments of the two recursive calls are sublists of smaller and greater elements w.r.t. a pivot element. These sublists themselves are computed by recursive partitioning functions.

The operator χ_{call} is based on matching the example outputs of the current unfinished rule for f with outputs belonging to (other) target or background functions f' and then computing arguments that appropriately map the inputs covered by the current rule for f to the corresponding inputs of f'.

Consider the unfinished initial rule for `sub` as introduced in the previous step: `sub((o : os), s) → s'`. The RHSs of both example rules of `sub` are subsumed by RHSs of the `rocket` examples. Particularly, `move(s)` (the RHS of the 1st `rocket` example) subsumes `move(load(o1, s))` (the RHS of the 1st `sub` example) by substitution $\sigma_1 = \{$`s` $\mapsto$ `load(o1, s)`$\}$; and `unload(o1, move(load(o1, s)))` (2nd `rocket` example) subsumes `unload(o2, move(load(o2, load(o1, s))))` (2nd `sub` example) by substitution $\sigma_2 = \{$o1 $\mapsto$ o2, s $\mapsto$ load(o1, s)$\}$. This indicates that the sub examples can be computed by calling rocket and the unfinished sub rule is refined to

```
sub((o : os), s) → rocket(g1((o : os), s), g2((o : os), s)) .
```

It remains to derive example rules for the new subfunctions g1 and g2 and computing initial rules for them. The example inputs are the same as for sub because g1 and g2 are called with the same inputs as sub, due to the same arguments o : os, s. The functions g1, g2 need to map these inputs to the correct inputs of rocket. Therefore, χ_{call} applies the substitutions σ_1, σ_2 to the *LHSs* of the rocket example rules and takes the appropriate subterms as outputs for g1, g2:

```
g1((o1 : nil), s)       → nil          g2((o1 : nil), s)       → load(o1, s)
g1((o1 : o2 : nil), s)  → o2 : nil     g2((o1 : o2 : nil), s)  → load(o1, s)
```

The initial rules (LGGs) for g1 and g2 obtained from their example rules are finished so that the following finished CS has been achieved as solution:

```
rocket(nil, s)        → move(s)
rocket((o : os), s)   → unload(o, sub((o : os), s))
sub((o : os), s)      → rocket(g1((o : os), s), g2((o : os), s))
g1((o : os), s)       → os
g2((o : os), s)       → load(o, s)
```

Neither sub nor g1, g2 are recursive. Hence they can be eliminated by unfolding, leading to the solution in Listing 1.2.

Like χ_{sub}, also χ_{call} does not increase the cost of the candidate because the added rules do not introduce additional patterns. To assure termination of IGOR2, the maximal depth of nested function calls, i.e., the maximal number of χ_{call} applications, is bounded by the user.

3 Discussion and Improvements

In this section we identify certain shortcomings of the splitting and the function call operators χ_{split} and χ_{call} and propose variants that circumvent the problems.

3.1 Rapid Rule-Splitting

Consider the *Ackermann* function, defined as a CS with constructors 0 (zero), S_ (successor) and variables m, n:

```
Ack (0, n)      → S n
Ack (S m, 0)    → Ack (m, S 0)
Ack (S m, S n)  → Ack (m, A (S m, n))
```

Given some I/O examples where all the four cases of zero and non-zero inputs for both arguments are covered, the initial unfinished rule would be Ack(m,n) → (S x), featuring two pivot positions that correspond to the variables m, n in the LHS. χ_{split} would thus introduce two successor candidates, each specializing one of the two pattern variables to the two cases zero and non-zero. W.l.o.g., let P denote one of them. P is unfinished again and does not contain the pattern of the *third* rule of the *Ackermann* CS because in that rule, *both* pattern components are non-variables. Thus, a further application of χ_{split} to P, leading, say, to P', where the cost of P' is increased compared to P, would be necessary. However, the subprogram and the function call operators would also be applicable to P *without* increasing its cost. Hence, before P' is considered again, all possible sequences of χ_{sub} and χ_{call} applications to P would be tried.

The idea of an improved version of χ_{split} is to *combine* all possible splitting refinements—if more than one pivot position and hence more than one splitting exists—into one single splitting. Instead of computing a separate partition for each pivot position, we compute only *one* partition based on all combinations of different constructors at all pivot positions. In the case of the *Ackermann* function, instead of two refinements with two successor rules each, we then get *one* refinement with *four* successor rules, covering all the four combinations for zero and non-zero inputs for the two arguments: Ack(0, 0), Ack(0, S n), Ack(S m, 0), Ack(S m, S n). These patterns cover all patterns of the actual definition of the *Ackermann* function such that subsequent applications of χ_{sub} and χ_{call} take place in a search subspace which contains the solution. Since this rule splitting variant achieves the result of several applications of χ_{split} in one step, we call it *rapid rule-splitting* and denote it by χ_{rsplit}. The solution for the Ackermann function that is induced with rapid rule-splitting enabled, is:

```
Ack (0, 0)     → S 0
Ack (0, S n)   → S S n
Ack (S m, 0)   → Ack (m, S 0)
Ack (S m, S n) → Ack (m, Ack (S m, n))
```

A minor drawback of rapid rule-splitting is its potential "over-specialization" as in the case of the *Ackermann* function (four induced rules instead of the sufficient three rules). This is not a problem as long as enough examples are provided. If, however, only few I/O examples are provided, then rapid rule-splitting might prevent a correct generalization since too few I/O examples might remain for each rule.

3.2 Simple Function Calls

Consider the following example rules, specifying the last function that returns the last element of a list (x,y,z denote variables):

```
1 last (x : nil)             → x
2 last (x : y : nil)         → y
3 last (x : y : z : nil)     → z
4 last (x : y : z : v : nil) → v
```

Further assume rule-splitting had already taken place so that the intermediate, unfinished, candidate CS is:

```
last (x : nil)      → x
last (x : y : xs)   → q
```

The second unfinished rule covers example rules $2, 3, 4$. Now assume we apply χ_{call} to introduce a recursive call of the form $\mathsf{last}\,(\mathsf{x}\ :\ \mathsf{xs}) \rightarrow \mathsf{last}\,(\mathsf{g}(\mathsf{x}\ :\ \mathsf{xs}))$. This is possible since each RHS of rules $2, 3, 4$ matches with another RHS. Actually, since all RHSs are variables, each RHS matches each other. Not all of these matchings are considered because, to assure termination of the induced program, the argument of the call must be decreased. Hence for each example i only matchings to examples $j < i$ are considered. One single refinement according to χ_{call} is then determined by one particular mapping of each RHS of rules $2, 3, 4$ to another one, satisfying the ordering constraint. In our case these are $1 \times 2 \times 3 = 6$ possibilities, hence χ_{call} would result in 6 successor candidates. In general, the more example rules are given, the more different matchings are possible and the more successor candidates are introduced by χ_{call}.

However, in the case of last, the argument of the recursive call need not be computed by an own function but is a constructor term, namely the tail of the input list: $\mathsf{last}\,(\mathsf{x}\ :\ \mathsf{y}\ :\ \mathsf{xs}) \rightarrow \mathsf{last}\,(\mathsf{y}\ :\ \mathsf{xs})$. If only candidates with a constructor term as argument are considered, the correct solution is *the only possible one*.

Therefore, we developed an additional operator, χ_{scall}, called simple-call operator, that finds refinements of the form $f(\boldsymbol{p}) \rightarrow f'(\boldsymbol{p'})$, where $\boldsymbol{p'}$ is a constructor term over variables from $\boldsymbol{p}$. Instead of matching specified outputs, χ_{scall} basically works by enumerating constructor terms as arguments up to a certain size and testing each one against the examples.

We did not completely *replace* χ_{call} by χ_{scall} but we always first apply χ_{scall} and only if it returns an empty refinement set—indicating, that a constructor argument is not sufficient to find a consistent function call—the original call operator χ_{call} is applied. The general idea is thus to first check for the few potential simple solutions and only if none exists, search for more complicated solutions.

4 Experiments

We implemented the extended synthesis operators on top of a preliminary IGOR2 version [8] that is implemented in the interpreted, rewriting based language MAUDE [2]. We use the symbol IGOR2*pre* to exclusively denote this version. To empirically evaluate the extensions, we applied IGOR2 to several non-trivial recursive problems. Each problem was tested with three IGOR2 configurations: (i) with IGOR2*pre*, (ii) with the simple-call operator χ_{scall} added as described in Sec. 3.2, and (iii) with χ_{scall} added and additionally with χ_{split} replaced by χ_{rsplit} (rapid rule-splitting), described in Sec. 3.1. The experiments were run on an Intel Core i5 2.53 GHz, 64Bit Linux machine. We gave each tested IGOR2 configuration 2 minutes synthesis time maximum per problem. Tab. 1 shows the results.

The *Blocksworld* problem tower is taken from [15] where IGOR2*pre* had been applied in the domain of cognitive modeling. tower denotes the recursive problem of building a tower of any number of blocks from initial configurations in the *Blocksworld*. The concept CB of how to clear a block (i.e., putting all blocks above

Table 1. Results

	Igor2 versions; times in sec.		
Problems	*pre*	$+\chi_{\text{scall}}$	$+\chi_{\text{scall}} + \chi_{\text{rsplit}}$
tower w/ CB, isTower	0.90	0.94	0.89
lasts	38.37	0.43	$0.54^{\perp}$
lasts w/ last	⊘	0.23	0.23
drop	⊘	9.87	0.05
swap	⊘	⊘	1.15
ack	⊘	⊘	2.22
weave	⊘	⊘	30.01
oddslist	⊘	⊘	⊘

⊘: Timeout after 2 minutes; $\perp$: one case overly special

pre: Igor2*pre*; $+\chi_{\text{scall}}$: χ_{call} only applied if χ_{scall} fails
$+\chi_{\text{scall}} + \chi_{\text{rsplit}}$: like $+\chi_{\text{scall}}$ and χ_{split} replaced by χ_{rsplit}

it to the table) was given as background knowledge as well as a predicate isTower to test if a certain tower is already present. We used the original examples. Since this problem is highly structured and the examples were well-chosen, Igor2*pre* could tackle it well and the extensions have no impact.

The problems lasts and oddslist are taken from [6] where Igor2*pre* has been compared with other recent IP systems on some list-processing problems. It was shown that Igor2, pursuing a combined analytic and search-based approach, (i) could correctly induce more problems than the recent analytic system Igor1 [9] and *inductive logic programming* systems like Foil [14] and (ii) outperformed recent generate-and-test based functional IP systems [12, 7] on several tested problems. lasts takes a list of lists and returns a flat list of their last elements. The predicate oddslist takes a list of natural numbers, encoded as Peano numbers by 0 and succ, and returns *true* or *false* depending on whether all elements are odd. No background knowledge was provided, so the IP systems had to invent subfunctions last and odd or equivalent ways to compute the inherent subproblems.

It is well-known in AI that background knowledge generally can help to find solutions for complex problems, but also that *irrelevant* information can hamper finding a solution. An odd thing with Igor2*pre* is that even *relevant* background knowledge may lead to increased synthesis time. This can be observed in the case of lasts which we tested (i) w/o background knowledge and (ii) with last as background knowledge. Igor2*pre* could not find a solution in 2 minutes if last was provided. In contrast, Igor2 with the additional χ_{scall} operator could, as one should expect, profit from the relevant background knowledge. The additional use of rapid rule-splitting had no further impact. We further observe that rapid rule-splitting did not completely generalize to the intended function in case of lasts w/o background knowledge. This is an example for that rapid rule-splitting might need additional examples to generalize well (cp. Sec. 3.1). The predicate

Listing 1.3. swap, induced from 6 examples

```
swap(x0 : x1 : xs, 0, 1)            → x1 : x0 : xs
swap(x0 : x1 : x2 : xs, 0, n+2)     → swap(x1 : swap(x0 : x2 : xs, 0, n+1), 0, 1)
swap(x0 : x1 : x2 : xs, n+1, m+2) → x0 : swap(x1 : x2 : xs, n, m+1)
```

Listing 1.4. weave, induced from 11 expls; note the automatically invented recursive subfunction sub36 that drops the first element of the first list and rotates the lists

```
weave(nil)                           → []
weave((x:xs) :: xss)                 → x : weave(sub36((x:xs) :: xss))
sub36((x:[]) :: nil)                 → nil
sub36((x:[]) :: (y:ys) :: xss)       → (y:ys) :: xss
sub36((x:y:xs) :: nil)               → (y:xs) :: nil
sub36((x:y:xs) :: (y:ys) :: xss)     → (y:ys) :: sub36((x:y:xs) :: xss)
```

oddslist could not be synthesized by any version within the allowed 2 minutes. Boolean-valued functions are generally hard for IGOR2 because of the missing structure in the outputs (which are just *true* or *false* in this case).

The function drop drops the first n elements from a list. We made this problem challenging for IGOR2*pre* by including the case where n is greater than the number of elements in the list in which case the empty list shall be returned. This leads to several I/O examples where the output is just the empty list, posing a problem for IGOR2*pre* because this causes many possible matchings of outputs. Since the solution does not contain nested functions calls, χ_{scall} quickly found a solution.

Finally, the *Ackermann* function ack and the functions weave and swap all are more complex than the former functions in terms of syntactical size, recursion structure, and/or number of parameters that are substituted in the recursive calls. They were neither solvable by IGOR2*pre* nor by just adding χ_{scall}. Yet with rapid rule-splitting enabled, many small candidates, mostly non-solutions, are pruned so that all three functions could be induced. swap swaps two elements in a list, indicated by their indices, e.g., swap([a,b,c,d], 2, 4) → [a,d,c,b]. It was restricted to cases where the given indices occurred in the list, were different, and the first index was the smaller one. Listing 1.3 shows the induced solution. weave takes a list of lists and produces a (flat) list by taking, in rotation over the inner lists, one element after the other from the inner lists.[2] Listing 1.4 shows the induced solution.

5 Related Research

Two recent functional IP systems are ADATE [12] and MAGICHASKELLER [7]. Both pursue a generate-and-test approach, i.e., use examples as test-cases, in-

[2] This is a generalized version of the weave function as tested in [6].

stead of directly deriving candidates from them as IGOR2 does. One advantage of this approach is that it is more robust w.r.t. the selection of and noise in problem specifications. However, they often need much more time to synthesize a solution [6]. In logic programming, the IP system DIALOGS [3] is closest to IGOR2. It is interactive and uses algorithm schemas like *divide-and-conquer*. Recently, domain-specific IP methods are again studied; e.g., [4] describes an algorithm to interactively synthesize string-processing programs in spreadsheets, and [11] describes a system to learn recursive hierarchical task networks in automated planning.

6 Conclusions and Future Work

IP is a challenging field with various important applications and much room for further improvement. We presented IGOR2, a competitive IP system that draws from different existing approaches to approach the practical tractability of relevant problems. We described improvements of two synthesis operators and empirically showed their significance w.r.t. efficient synthesis of non-trivial programs. It is worth noting that the efficiency-gain is *not* based on making the search less complete. The only drawback is that more examples might be needed in case of rapid rule-splitting. Despite the remarkable results, there is still much room for improvement. The current synthesis operators rule out certain program forms and if the examples do not contain sufficient structure, the BF search becomes intractable.

One serious disadvantage of analytical techniques including IGOR2 is the requirement for sets of I/O examples that are complete up to some complexity. This often compels the programmer/specifier to think about missing I/O pairs even if a meaningful I/O pair, that would suffice for a generate-and-test method, is already provided. On the logical side, this problem could be tackled by introducing and reasoning with ∃-quantified variables in example outputs. To generally become more robust w.r.t. missing or erroneous information, as generally present in the real-world, probabilistic reasoning must be integrated into analytic IP.

Currently, we work on applying IGOR2 to learning hierarchical task networks in automated planning.

Acknowledgments

While writing this paper, the author was funded by the German Academic Exchange Service (DAAD) within the FIT program.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1999)

2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Rewriting Techniques and Applications (RTA'03). LNCS, vol. 2706, pp. 76–87. Springer (2003)
3. Flener, P.: Inductive logic program synthesis with DIALOGS. In: 6th International Workshop on Inductive Logic Programming, (ILP'96), Selected Papers. LNCS, vol. 1314, pp. 175–198 (1997)
4. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: 38th SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM (2011)
5. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer-Verlag, 2nd edn. (2009)
6. Hofmann, M., Kitzelmann, E., Schmid, U.: A unifying framework for analysis and evaluation of inductive programming systems. In: Artificial General Intelligence (AGI'09). pp. 55–60. Atlantis Press (2009)
7. Katayama, S.: Systematic search for lambda expressions. In: 6th Symposium on Trends in Functional Programming, selected Papers. pp. 111–126. Intellect (2007)
8. Kitzelmann, E.: Analytical inductive functional programming. In: 18th International Symposium on Logic-Based Program Synthesis and Transformation, Revised Selected Papers. LNCS, vol. 5438, pp. 87–102. Springer-Verlag (2009)
9. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7, 429–454 (2006)
10. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press (1969)
11. Nejati, N., Langley, P., Konik, T.: Learning hierarchical task networks by observation. In: 23rd International Conference on Machine Learning. pp. 665–672. ACM (2006)
12. Olsson, J.R.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–83 (1995)
13. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1970)
14. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Proceedings of the 6th European Conference on Machine Learning. pp. 3–20. LNCS, Springer-Verlag (1993)
15. Schmid, U., Kitzelmann, E.: Inductive rule learning on the knowledge level. Cognitive Systems Research (2010)
16. Smith, D.R.: The synthesis of LISP programs from examples: A survey. In: Automatic Program Construction Techniques, pp. 307–324. Macmillan (1984)
17. Veloso, M.M., Carbonell, J.G.: Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. Machine Learning 10, 249–278 (1993)

MagicHaskeller: System demonstration

Susumu Katayama

University of Miyazaki
1-1 W. Gakuenkibanadai, Miyazaki, Miyazaki 889-2192, Japan
skata@cs.miyazaki-u.ac.jp

Abstract. This short paper introduces the usage and behavior of MagicHaskeller , which is one of the representative inductive functional programming systems. Although MagicHaskeller had been a generate-and-test method based on systematic exhaustive search, an analytical synthesis engine was added to its recent versions, which enables a new method that generates many programs analytically from the given insufficient set of input-output examples and tests those programs with a separately given predicate. This paper mentions both engines.

1 Overview

MagicHaskeller [Katayama(2005b)] is an inductive functional programming system based on systematic exhaustive search. *Inductive functional programming (IFP)* is a form of programming automation, where recursive functional programs are synthesized through generalization from the ambiguous specification usually given as a set of input-output pairs. Currently, there are two approaches to IFP: *analytical approach* that synthesize programs by looking into the input-output pairs and conducting inductive inference, and *generate-and-test approach* that generates many programs and picks up those that satisfy the specification. MagicHaskeller has been playing the representative role as the generate-and-test method based on systematic exhaustive search since the first binary release in 2005. Since its Version 0.8.6 release, analytical search algorithm has been added, and a new approach that could be called *analytically-generate-and-test approach* has been made possible, where the analytical synthesizer generates many programs from the given insufficient set of input-output examples and picks up those that satisfy the predicate separately given as a part of the specification. In this short demonstration paper, we present how to use both synthesis engines and the results from use of them.

2 Building and installation

MagicHaskeller has been developed in Haskell and its recent versions are released as a library. In order to build and install its copy, first you need to install Version 6.10.* or 6.12.* of Glasgow Haskell Compiler (GHC). Although Version 0.8.6.1 of MagicHaskeller can be build with Version 7 of GHC, its analytically-generate-and-test functionality does not work with this version of GHC.

Also, in order to ease the installation process you should have the Cabal [Jones(2005)] package that is the standard framework for distributing HASKELL programs. In addition, if cabal-install package is installed, simply typing

```
cabal update
cabal install MagicHaskeller
```

builds and installs the MAGICHASKELLER system into the user's home directory.

Implemented as a library, the typical usage of MAGICHASKELLER is to use it within Glasgow Haskell Compiler interactive (GHCi), like **QuickCheck** [Claessen and Hughes(2000)]. This is achieved by invoking GHCi with `-package MagicHaskeller` option. The language extension with Template Haskell [Sheard and Peyton Jones(2002)] is also necessary if you want to do various things by using its oxford bracket syntax. Thus, MAGICHASKELLER is usually invoked with

```
ghci -package MagicHaskeller -XTemplateHaskell
```

In this paper, when quoting use of the interactive system, we always supply the `-v0` option which means verbosity level 0, in order to avoid clutter.

3 The modules for exhaustive search

The systematic exhaustive search modules define functions that generate all the programs (up to semantical equivalence) which can be constructed using the given primitive set with function applications and lambda abstractions, as an infinite stream[Katayama(2005a)]. They also define functions for testing the generated programs to leave only the programs that satisfy the given specification. The algorithm used for generating the stream of programs effectively enumerates all the proofs for the proposition corresponding to the given type under Curry-Howard isomorphism, based on sequent calculus.[Katayama(2010)]

The greatest feature of these modules is that they can synthesize programs by only selecting the primitive set and writing the specification in the form of a predicate. The specification need not be a set of input-output pairs. The type of the function has to be notified to the algorithm, but the user need not give it explicitly. It can be inferred from the specification given as a predicate.

To the end of this section, we exemplify the usage of the systematic exhaustive search engine. More examples can be found in [Katayama(2006)], though it describes an older version of MAGICHASKELLER .

3.1 A simple example

This is an example of having MAGICHASKELLER synthesize functions that takes `"abc"` and returns `"aabbcc"`:

```
$ ghci -package MagicHaskeller -XTemplateHaskell -v0
Prelude> :m +MagicHaskeller.LibTH
Prelude MagicHaskeller.LibTH> init075
Prelude MagicHaskeller.LibTH> printAll $ \f -> f "abc" == "aabbcc"
\a -> list_para a [] (\b _ d -> b : (b : d))
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d 0)) 0
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d 0)) 0
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d True)) True
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d True)) False
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d False)) True
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d False)) False
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d [])) []
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d [])) a
\a -> list_para a (\_ -> []) (\b c d _ -> b : (b : d c)) a
\a -> list_para a (\_ -> []) (\b c d _ -> b : (b : d c)) []
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d Nothing)) Nothing
\a -> list_para a (\b -> b) (\b _ d e -> b : (b : d e)) []
\a -> list_para a (\b -> b) (\b c d _ -> b : (b : d c)) a
\a -> list_para a (\b -> b) (\b c d _ -> b : (b : d c)) []
\a -> list_para a (\b -> b) (\b _ d _ -> b : (b : d [])) a
\a -> list_para a (\b -> b) (\b _ d _ -> b : (b : d [])) []
^CInterrupted.
Prelude MagicHaskeller.LibTH> printAllF $ \f -> f "abc" == "aabbcc"
\a -> list_para a [] (\b _ d -> b : (b : d))
^CInterrupted.
```

The text regions between `Prelude` and `>` are prompts of GHCi. The first line after the GHCi invocation is for bringing module `MagicHaskeller.LibTH` into scope. The second line initializes the environment and set the component library, i.e. the set of combinators with which to construct programs, with a recommended set of combinators. The third line requests to print all the expressions which can be synthesized using the combinators in the component library that satisfy the predicate `\f -> f "abc" == "aabbcc"`.

Then, synthesized programs are printed line by line, from the smallest one with the least number of function applications, increasing the program size progressively. The `list_para` function is (a function isomorphic to) the list paramorphism (e.g. [Augusteijn(1999)]) and defined in the module `MagicHaskeller.LibTH`. Other notations are the same as HASKELL's — `\v -> e` means $\lambda v.e$, `[]` denotes the empty list, and `(:)` is the binary constructor that adds one element to the first position of a list. Because the algorithm generates an infinite stream of programs, it has to be interrupted on the way.

The letter `F` in `printAllF` means filtering out expressions that are semantically equivalent to any of the already printed expressions by using a randomized algorithm.[Katayama(2008)] This is useful, though it affects the efficiency. In the above example, seemingly all the synthesized and printed programs are equivalent to the firstly generated one. However, there is always the possibility where some of them are proved to be different and printed later while using `printAllF` if the computation is not interrupted.

3.2 An example of using a rich library

A variant of the filter for removing semantically equivalent expressions can be applied during program generation rather than after program generation, in order to reduce the number of programs and quicken the synthesis. Because the filtration itself takes time, program generation with this technique is not always quicker than that without it. However, it is known to be quicker when using a rich set of combinators as the component library. In the Version 0.8.6.1 of MagicHaskeller , it can be tried by using `MagicHaskeller.LibTH.exploit`.

```
Prelude MagicHaskeller.LibTH> exploit $ \f -> f "abc"=="abcba"
\a -> list_para (reverse a) a (\_ c _ -> a ++ c)
^CInterrupted.
```

Although use of a rich library may be considered as cheating when benchmarking, it can be more useful than using a poor library, making the results more readable.

4 The modules for analytical synthesis

Since Version 0.8.6, an analytical synthesis engine is added to MagicHaskeller . It uses an algorithm that extends Igor II [Kitzelmann(2007)] [Hofmann et al.(2010)Hofmann, Kitzelmann, and Schmid] to enable generation of many programs as a stream of lists, where programs with few case splittings are highly prioritized and appear early. This is made possible by not stopping search when the best programs are found but continuing the search in the breadth-first manner.

Igor II sometimes suffers from the trade-off between the correctness of the generated programs and the computational complexity: (only) unwanted programs are generated unless there are enough number of examples to correctly specify the desired function, and no programs are generated if the number of examples is too large due to the time complexity for conducting unification. This trade-off often forces trial-and-error to users in order to find the adequate number of I/O example pairs, and sometimes causes search failure due to lack in such a number. Those cases often happen especially when arguments increase in different dimensions and there are some corner cases.

For example, Igor II cannot correctly synthesize the `concat` function in the Standard Prelude of Haskell and the `allodd` function that takes a list of integers and returns if all of its elements are odd. In both cases wrong functions are generated when given several examples, and no functions are obtained, at least within five minutes, when given more than ten examples.

Generating many programs analytically from insufficient input-output pairs and then filtering them with a separately-supplied predicate is an answer to this trade-off. In fact, those functions can be synthesized with our new analytical synthesis modules without any trial-and-error on the users' side. The trade-off is resolved by dividing the set of input-output examples into those for guiding the search and those for avoiding generation of unintended solutions. For the latter purpose, the user usually need only one general (in that it is not at an edge or corner case) example, and rarely need to enumerate many examples. In addition, even if some examples are required here, that hardly influences the efficiency. On the other hand, the set of input-output examples for guiding the search can be minimized. Moreover, even if there are not enough number of examples for this purpose, the solution can be found, though the search is less efficient than when the optimal number of examples are given.

The advantage of analytical synthesis over systematic exhaustive search is that there are functions which the exhaustive algorithm cannot synthesize within a realistic time span but analytical algorithms can. On the other hand, some other functions such as the Fibonacci function can be synthesized by exhaustive search but cannot be synthesized by analytical algorithms (without using a specialized addition function, which can be regarded as cheating). Currently the analytical synthesis engine works separately from the systematic exhaustive search engine, except that they share some common modules such as those defining the language and those implementing combinatorial search. Their co-operation to make a new synthesis engine will be tried in future. Also note that a paramorphism introduction operator like in
[Hofmann and Kitzelmann(2010)] is not implemented yet.

4.1 A simple example

In the next example the length function is synthesized. The analytical synthesis engine generates many programs that generalize `{f [] = 0; f [a] = 1}` without using any background knowledge functions. Then, it compiles the generated programs, and filter them with the predicate `\f -> f "12345" == 5`.

```
$ ghci -package MagicHaskeller -XTemplateHaskell -v0
Prelude> :m MagicHaskeller.RunAnalytical
Prelude MagicHaskeller.RunAnalytical> :set prompt >
> quickStart [d| f [] = 0; f [a] = 1 |] noBKQ (\f -> f "12345" == 5)
\a -> let fa (b@([])) = 0
          fa (b@(c : d)) = succ (fa d)
       in fa a :: forall t2 . [t2] -> Int
^CInterrupted.
```

The first two lines are not essential. The first line is for bringing module `MagicHaskeller.RunAnalytical` into scope. The second line literally replaces the command prompt with `>>`. The latter is not indispensable, but it is recommended when using a narrow screen, because a lot of information has to be input at each analytical synthesis.

The third line is important, though is not very difficult. The set of declarations surrounded by the Oxford bracket `[d| . |]` is a declaration quote of Template Haskell having type `Q [Dec]`. The `quickStart` function takes the set of input-output pairs of the target function as the first argument, the sets of input-output pairs of the background knowledge functions, and the predicate with which to filter the generated programs. The careful reader may notice that different syntaxes are used between the first argument and the third argument. For example, a variable pattern is used in the first argument while concrete values are used in the third argument, and the implicit equality is used in the first argument while the explicit one is used in the third argument. These are not the results of the author's fancy. Variable patterns can be used within the first argument because it has the form of a function definition. They are often required by analytical synthesis in order to make recursive calls, because a recursive call involves pattern matching. On the other hand, the third argument that is used for filtering the generated programs has to use concrete values, because they will not be abstractly interpreted but compiled and executed.

Although in the above example the type of the target function is not supplied, it may be supplied as the type signature declaration in the first argument, like `[d| f :: [a]->Int; f [] = 0; f [a] = 1 |]`. When the type signature is omitted, the type is inferred from the types of constructors appearing in the input-output pairs. Integral literals are assumed to have type `Int`. They are treated specially and converted into combinations of `0`, `succ`, and `negate`.

Patterns with `@` are called *as-pattern*s, and the argument is matched to both patterns at the both sides of `@`. In the above case, because the two `b`s are unused, use of as-patterns are actually unnecessary. Such redundant use of as-patterns will be removed from the future releases.

4.2 An example with a background knowledge function

When background knowledge function(s) should be used, they are specified in the second argument. In this case, the analytical synthesizer generates higher-order functions that take background knowledge functions as arguments. This should be noted when specifying the test function.

The next example shows synthesis of multiplication using addition as the background knowledge function. Note that multiplication is one of the boring functions that cannot be synthesized by IGOR II. Since `(+) :: Int -> Int -> Int` is used as the background knowledge function, the resulting programs require `(+)` as the first argument.

```
> :{
| quickStartF
|     [d| mult 0 x = 0;    mult 1 0 = 0;
|         mult 1 1 = 1;    mult 1 2 = 2;
|         mult 2 0 = 0;    mult 2 1 = 2;
|         mult 2 2 = 4 |]
|     [d| add 0 x = x;     add 1 0 = 1;
|         add 1 1 = 2;     add 1 2 = 3;
|         add 2 0 = 2;     add 2 1 = 3;
|         add 2 2 = 4 |]
|     (\f -> f (+) 5 6 == 30)
| :}
\fa a b -> let fb (c@0) d = 0
               fb (c@succe) d | succe > 0 = fa d (fb e d)
                      where e = succe - 1
```

```
            in fb a b :: (Int -> Int -> Int) -> Int -> Int -> Int
\fa a b -> let fb (c@0) d = 0
               fb (c@succe) d | succe > 0 = fa (fb e d) d
                     where e = succe - 1
            in fb a b :: (Int -> Int -> Int) -> Int -> Int -> Int
^CInterrupted.
```

`:{` and `:}` are used in order to input a multi-line expression. This syntax of GHCi is introduced to Version 7, and module `MagicHaskeller.RunAnalytical` does not work with the version of GHCi, but the actual one-line input is hand-edited into this syntax in order to fit the input line into the page width of this paper. Also, the effect of the command for changing the prompts is actually cancelled within the `:{ . :}` block, but we assume that it works there.

In this example, we used an action with letter `F`, namely `quickStartF`, in order to avoid printing equivalent functions. Of course, we could use `quickStart` here instead, though doing so in this case results in printing tons of expressions. The two results printed are still equivalent if we limit the background knowledge function to `(+)`, but they are different if we use a non-commutative operator. For this reason, these two functions are recognized as different.

4.3 Using types unknown to MagicHaskeller

The actions introduced so far only works for types known beforehand, whose constructors appear in `MagicHaskeller.CoreLang.defaultPrimitives`. Types that do not appear there can be dealt with in the following way:

```
> :{
| quickStartCF $(c [d| f [] = 0; f [a] = 1; f [a,b] = 2 |]) noBK
|               (\f -> f "foobar" == 6)
| :}
\a -> let fa (b@([])) = 0
          fa (b@(c : d)) = succ (fa d)
       in fa a :: forall t6 . [t6] -> Int
^CInterrupted.
```

where the function `c` extracts the values of constructors appearing in the Oxford bracket and hand them over to the `quickStartCF` action. The code block between `$(` and `)` describes what should be spliced, and there must not be spaces between `$` and `(`.

5 Conclusions

This paper exemplified the usage and behavior of the newest version of MagicHaskeller. In addition to the usage of its older systematic exhaustive search engine, that of the newly added analytical synthesis engine was also explained. The new analytical synthesis engine is based on Igor II, but can generate many hypothesis programs. By generating many programs analytically and testing them, we can have a new synthesis system that is more powerful than Igor II.

References

[Augusteijn(1999)] L. Augusteijn. Sorting morphisms. In *Advanced Functional Programming, LNCS 1608*, pages 1–27. Springer Verlag, 1999.

[Claessen and Hughes(2000)] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP'00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[Hofmann and Kitzelmann(2010)] Martin Hofmann and Emanuel Kitzelmann. I/O guided detection of list catamorphisms: towards problem specific use of program templates in ip. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 93–100, 2010.

[Hofmann et al.(2010)Hofmann, Kitzelmann, and Schmid] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Porting IgorII from maude to haskell. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009*, volume 5812 of *LNCS*, pages 140–158, 2010.

[Jones(2005)] Isaac Jones. The haskell cabal: A common architecture for building applications and libraries. In *Sixth Symposium on Trends in Functional Programming*, pages 340–354, 2005.

[Katayama(2005a)] Susumu Katayama. Systematic search for lambda expressions. In *Sixth Symposium on Trends in Functional Programming*, pages 195–205, 2005a.

[Katayama(2005b)] Susumu Katayama. MagicHaskeller: A search-based inductive functional programming system. http://nautilus.cs.miyazaki-u.ac.jp/˜skata/MagicHaskeller.html, 2005b.

[Katayama(2006)] Susumu Katayama. Library for systematic search for expressions and its efficiency evaluation. *WSEAS Transactions on Computers*, 12(5):3146–3153, 2006.

[Katayama(2008)] Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In Tu Bao Ho and Zhi-Hua Zhou, editors, *PRICAI*, volume 5351 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2008. ISBN 978-3-540-89196-3.

[Katayama(2010)] Susumu Katayama. Recent improvements of MagicHaskeller. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009*, volume 5812 of *LNCS*, pages 174–193, 2010.

[Kitzelmann(2007)] Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *AAIP'07: Proceedings of the Workshop on Approaches and Applications of Inductive Programming*, pages 15–26, 2007.

[Sheard and Peyton Jones(2002)] Tim Sheard and Simon L. Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop 2002*, October 2002. URL http://research.microsoft.com/Users/simonpj/papers/meta-haskell/meta-haskell.ps.