Automatic Invention of Functional Abstractions

Robert J. Henderson and Stephen H. Muggleton

Department of Computing, Imperial College London, United Kingdom
{rjh09,shm}@doc.ic.ac.uk

Abstract. We investigate how new elements of background knowledge can be abstracted automatically from patterns in programs. The approach is implemented in the *KANDINSKY* system using an algorithm that searches for common subterms over sets of functional programs. We demonstrate that KANDINSKY can invent higher-order functions such as *map*, *fold*, and *sumBy* from small sets of input programs. An experiment shows that KANDINSKY can find high-compression abstractions efficiently, with low settings of its input parameters. Finally we compare our approach with related work in the inductive logic programming and functional programming literature, and suggest directions for further work.

1 Introduction

Can background knowledge be learned automatically through problem-solving experience? This would be a form of *meta-learning* [7], distinct from *base learning* which is concerned simply with solving problem instances. We propose that a general strategy for acquiring new background knowledge can be found in the *abstraction principle* of software engineering. *Abstractions* [1] are re-usable units obtained by separating out and encapsulating patterns in programs. We define *abstraction invention* as the process of formulating useful abstractions in an inductive programming context, and when these abstractions take the form of functions, *Functional Abstraction Invention* (FAI). Some forms of *predicate invention* may be regarded as FAI (since predicates are functions).

We have implemented *KANDINSKY* [1], a system which performs FAI over sets of functional (λ-calculus) programs by *Inverse β-Reduction* (IβR), an analogue of inverse resolution [4, 5]. This move from first-order logic to λ-calculus is crucial because it allows our system to invent *higher-order* functional abstractions, an ability that is necessary in order to generalise on arbitrary patterns in programs. See Fig. 1 for an example where first-order methods fail.

```
incElems([],[]).                      doubleElems([],[]).
incElems([H|T],[H1|T1]) :-            doubleElems([H|T],[H1|T1]) :-
    inc(H,H1), incElems(T,T1).            times(2,H,H1), doubleElems(T,T1).
```

Fig. 1. To abstract over the commonality manifest in the above two programs requires quantification over predicate symbols, which is impossible in first-order logic.

[1] Source code available at: http://ilp.doc.ic.ac.uk/kandinsky

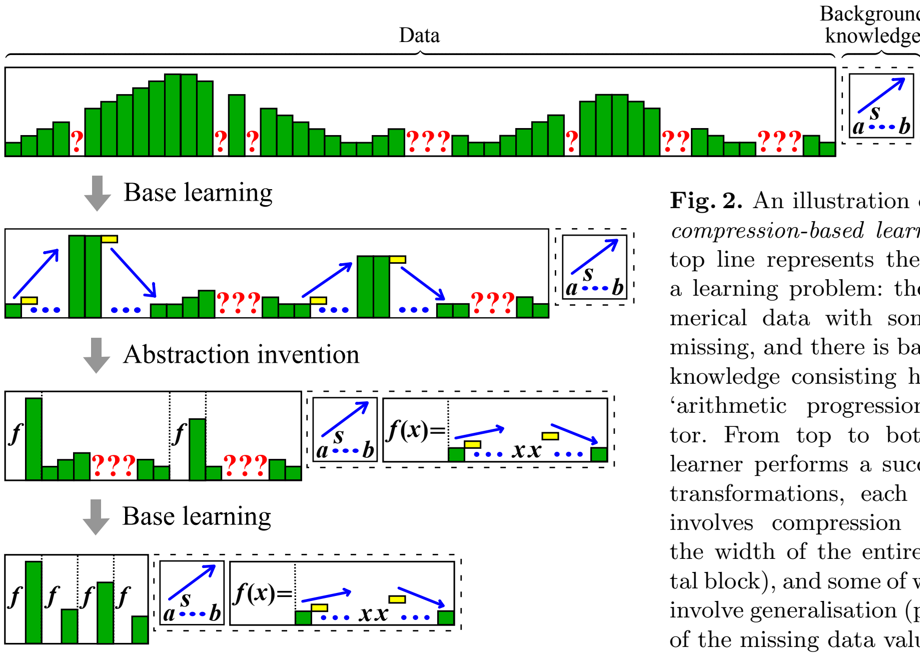


Fig. 2. An illustration of *compression-based learning*. The top line represents the input in a learning problem: there is numerical data with some values missing, and there is background knowledge consisting here of an 'arithmetic progression' operator. From top to bottom, the learner performs a succession of transformations, each of which involves compression (reducing the width of the entire horizontal block), and some of which also involve generalisation (prediction of the missing data values).

KANDINSKY is set within a larger inductive programming framework called Compression-Based Learning (CBL). CBL takes advantage of the general correspondence that exists between learning and compression (*minimum description length* [3]), to allow both base learning and meta-learning to be understood in a unified manner in terms of transformation operators. See Fig. 2 for an illustration of CBL. We shall leave further discussion of CBL for a future paper; the rest of this paper is concerned only with the FAI meta-learning technique.

2 KANDINSKY's Abstraction Invention Algorithm

Given a set of k λ-calculus terms, each with at most n subterms, the number of possible combinations of two or more subterms with at most one subterm taken per term is of the order of $(n+1)^k$. Any of these combinations can potentially be *anti-unified* by IβR to form an abstraction, but enumerating all of them by brute force is intractable for even moderately large values of k and n. To cope with this, we have designed a heuristic search procedure *auSearch* (anti-unification search) whose running time is polynomial in both k and n.

To prepare a set of terms for *auSearch*, all their subterms are generated, converted to a tree representation (Defn. 1), and each subterm paired with a 'tag' (Defn. 2) marking its origin. *auSearch* itself (Fig. 3) searches the space of 'common parts' (Defn. 1) that are obtainable by anti-unifying combinations of two or more subterms. It makes use of a heuristic 'score' function (Defn. 3) in order to guide the search. Each *auSearch* result represents one candidate

Input: $\sigma, \tau \in \mathbb{N}$; I, a set of $\langle \text{tag}, \text{tree} \rangle$ pairs.
Output: a set of *auSearch* results.

1. **proc** *auSearch*(σ, τ, I):
2. Let $R = \mathit{bagof}(\mathit{findOne}(\sigma, \tau, I))$.
3. Divide R into disjoint subsets by tag-set. For each of these subsets, extract the top σ elements by score. Place all of the extracted elements into a new set R'.
4. Return the top τ elements by score of R'.

Fig. 3. The *auSearch* algorithm. *findOne* is a non-deterministic procedure that returns many *auSearch* results on backtracking. *findOne* and *auSearch* are mutually recursive. Owing to lack of space we defer a specification of *findOne* to a longer paper.

abstraction, which can be constructed from the subterms marked by the result's tag-set.

Definition 1 (tree, node, common part, mismatch point, size). *A* tree *is a pair $\langle h, B \rangle$ where h is a* node *and B is a list of trees. In the representation of λ-terms as trees, each node is a symbol representing either a variable, a function application, or an anonymous function (λ-abstraction). A* common part *is either a* mismatch point $\bullet$*, or a pair $\langle h, B \rangle$ where h is a node and B is a list of common parts. The* size *of a tree or common part is equal to the number of nodes it contains. A mismatch point has zero size.*

Definition 2 (tag, term index, subterm index). *A* tag *consists of a pair of integers called the* term index *and the* subterm index*. It represents a reference to a particular subterm within a particular term, given a list of terms.*

Definition 3 (*auSearch* result, score). *An auSearch* result *is a pair of the form $\langle \gamma, T \rangle$, where γ is a common part and T is a tag-set (set of tags). Its* score*, an approximation to the degree of compression that can be obtained by deriving an abstraction from this result, is given by $(n-1)c - (n+2)m - n$, where n is the number of unique term indices contained in T, c is the size of γ, and m is the number of mismatch points contained in γ.*

auSearch has two *beam size* parameters σ and τ which limit how many intermediate results are stored during the search. When these parameters are both infinite, the search is complete but has exponential time complexity in the size of the input; when they are finite, the search is incomplete but the time complexity is polynomial.

Equipped with *auSearch*, KANDINSKY can perform a process called *exhaustive greedy abstraction invention*. Here, a set of programs is provided as input, and KANDINSKY constructs the most compressive abstraction that it can find, adds it to the set, and re-expresses the other programs in terms of it. This process repeats continually, halting only when no further compression is possible. A demonstration on two (hand-constructed) datasets *Map-Fold* (MF) and *Sum-Means-Squares* (SMS) is shown in Fig. 4.

a). Map-Fold dataset (size = 71)

```
incElems    = fix (\ r lst -> if (null lst) nil (cons
                (inc (head lst)) (r (tail lst))))
doubleElems = fix (\ r lst -> if (null lst) nil (cons
                (times two (head lst)) (r (tail lst))))
length      = fix (\ r lst -> if (null lst) zero (inc
                (r (tail lst))))
```

↓

After 1 stage (size = 53, compression = 25.4%)

```
g1          = \ a1 -> fix (\ a2 a3 -> if (null a3) nil
                (cons (a1 (head a3)) (a2 (tail a3))))
incElems    = g1 inc
doubleElems = g1 (times two)
length      = fix (\ a4 a5 -> if (null a5) zero (inc
                (a4 (tail a5))))
```

↓

After 2 stages (size = 50, compression = 29.6%)

```
g2          = \ a1 a2 -> fix (\ a3 a4 -> if (null a4)
                a1 (a2 a4 (a3 (tail a4))))
g1          = \ a5 -> g2 nil (\ a6 -> cons (a5 (head
                a6)))
incElems    = g1 inc
doubleElems = g1 (times two)
length      = g2 zero (\ a7 -> inc)
```

b).

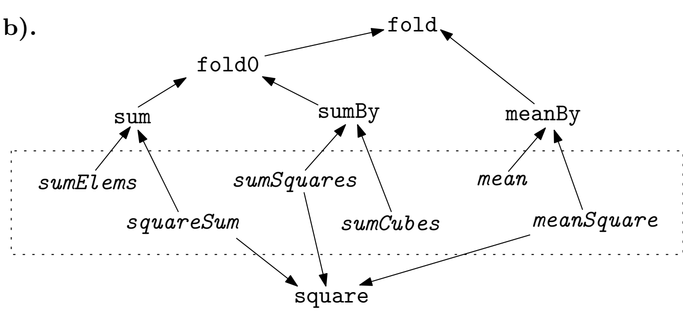


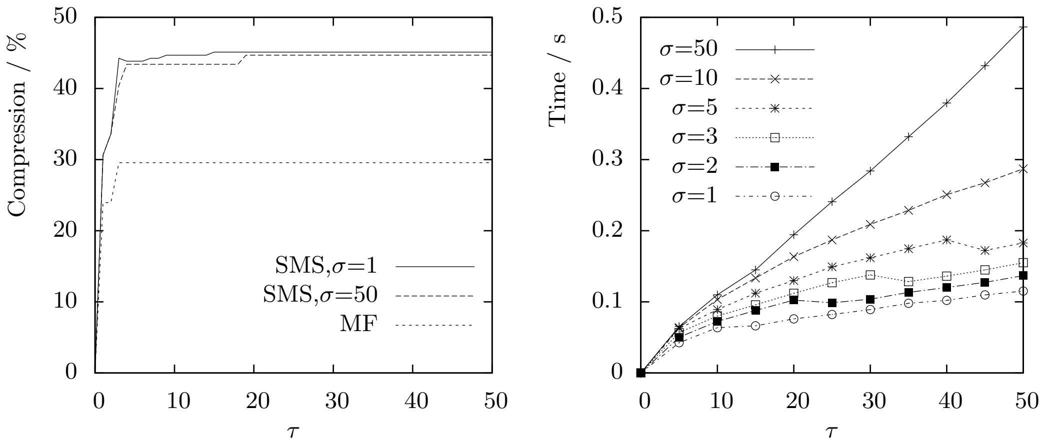
Fig. 4. a). KANDINSKY's output trace on the *Map-Fold* dataset, which consists of three list-processing programs. In stage 1, KANDINSKY antiunifies `incElems` with `doubleElems` to produce an abstraction `g1` which we may recognise as *map*, a higher-order function which maps an arbitrary unary operation over the elements of a list. In stage 2, KANDINSKY antiunifies `length` with a subterm of `g1` to produce `g2`, a form of *fold* which accumulates over a list using a binary operation. **b).** Summary of results for the *Sums-Means-Squares* dataset. The input programs (inside dashed rectangle) express various actions over the elements of a list. KANDINSKY succeeded in finding six abstractions, which we inspected and assigned suitable names. `sumBy` is a higher-order analogue of `sum` which maps an arbitrary function over a list before summing its elements; `meanBy` is a generalisation of `mean` along similar lines. `fold0` is a specialisation of `fold`.

3 Experiment

In this section we ask: *in practice, can we expect KANDINSKY to find near-optimally compressive abstractions in polynomial time?* As discussed in Sect. 2, *auSearch* can always find an optimally compressive abstraction when the beam size parameters σ and τ are infinite, because under those conditions it generates every abstraction in the entire search space. However, finite values of the beam size parameters are necessary for a tractable polynomial-time search. By studying the effect on compression of varying these parameters, we wish to determine if one can expect to achieve near-optimal compression even when using relatively small finite values.

a). Effect of τ on compression, for both datasets.

b). Effect of τ and σ on time taken, for SMS.



σ	1	2	3	5	10	50
τ	15	24	32	44	–	–

c). Effect of σ on the smallest value of τ at which the compression plateau of 45.1% was reached, for SMS.

Fig. 5. Experimental results. The two datasets are 'Map-Fold' (MF) and 'Sums-Means-Squares' (SMS). σ and τ are the beam-size parameters of KANDINSKY's search algorithm. In a), the compression-τ curves for MF are all identical for the six values of σ that were tested; for SMS they are all different but lie very close together, so we have only plotted those for the lowest and highest σ values. In b), we have plotted the curves for SMS only; the curves for MF show a somewhat similar pattern. The experiment was run on a 2.8 GHz desktop PC with 4 GB of RAM.

We ran exhaustive greedy abstraction invention on the MF and SMS datasets, at values of σ of 1, 2, 3, 5, 10, and 50, for all values of τ between 0 and 50, and recorded the overall compression for each run (Fig. 5a). We also measured the time taken at the same values of σ and for values of τ at $0, 5, 10 \ldots 50$ (Fig. 5b). To reduce the effects of measurement error, each timing measurement was averaged over 200 identical runs for MF and 20 identical runs for SMS.

From the results, we see that as tau increases, compression increases. However, the vast majority of compression is achieved for both datasets by $\tau = 4$: the compression curves reach a 'plateau' very rapidly. For larger values of σ, a larger value of τ tends to be needed to reach the maximum achievable level of compression (Fig. 5c). Time taken increases with both τ and σ.

The 'plateau' phenomenon that we observe supports the hypothesis that low beam size parameters are adequate for achieving near-optimal compression. For the datasets studied here, it seems unlikely that the plateau is a 'false-summit', because the invented abstractions capture almost all of the obvious commonality manifest in the input programs. However, whether this plateau effect will occur for arbitrary input programs is an open question; ultimately it would be worth trying to obtain a theoretical justification.

4 Related/Further Work and Conclusion

Our FAI technique is inspired by a standard 'recipe' which human programmers use to derive functional abstractions from patterns in programs, described by e.g. Abelson and Sussman [1, Sect. 1.3.1]. One previous attempt to automate this kind of recipe is due to Bakewell and Runciman [2]; they implemented an abstraction construction algorithm for Haskell programs, however they did not address the problem of searching for a compressive abstraction. In inductive logic programming, the Duce [4] and CIGOL [5] systems use inverse resolution to perform FAI in propositional and first-order logic respectively; KANDINKSY shares a lot with these systems, both its inverse deduction approach (IβR), as well as its use of a compression-guided search algorithm.

For further work, we hope shortly to combine KANDINSKY with a base learning system so as to realise a full CBL framework. Many improvements can also be made to KANDINSKY itself; most significantly it is currently limited to deriving abstractions from *syntactic* commonality in programs, whereas a more powerful system could search the space of all *semantic* equivalences via β-η-δ transformations.

To conclude, we have defined the term *abstraction invention* to mean the derivation of new knowledge from patterns in programs. We have demonstrated and experimentally justified an efficient algorithm for *functional* abstraction invention over λ-calculus programs in the KANDINSKY system. KANDINSKY invented, without any prior knowledge of such concepts, higher-order functions such `map`, `fold`, `sumBy`, and `meanBy`. Some of these functions are strikingly similar to ones in the Haskell standard library [6], so KANDINSKY is clearly able to invent abstractions that are natural from a human perspective.

Acknowledgments. Thank you to Dianhuan Lin, Alireza Tamaddoni-Nezhad, and Jianzhong Chen, for their helpful comments on an earlier draft of this paper.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Massachusetts, 2nd edn. (1996)
2. Bakewell, A., Runciman, C.: Automated generalisation of function definitions. In: FLOPS'99. LNCS, vol. 1722, pp. 225–240. Springer-Verlag (1999)
3. Grünwald, P.D., Myung, J., Pitt, M.A.: Advances in Minimum Description Length: Theory and Applications. MIT Press (2005)
4. Muggleton, S.: Duce, an oracle based approach to constructive induction. In: IJCAI-87. pp. 287–292 (1987)
5. Muggleton, S., Buntine, W.: Machine invention of first-order predicates by inverting resolution. In: Proceedings of the 5th International Conference on Machine Learning. pp. 339–352. Morgan Kaufmann (1988)
6. Peyton Jones, S.L.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
7. Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Artificial Intelligence Review 18(2), 77–95 (2002)