THE 18TH EUROPEAN CONFERENCE ON MACHINE LEARNING
AND
THE 11TH EUROPEAN CONFERENCE ON PRINCIPLES AND PRACTICE
OF KNOWLEDGE DISCOVERY IN DATABASES

PROCEEDINGS OF THE WORKSHOP ON APPROACHES AND APPLICATIONS OF INDUCTIVE PROGRAMMING

AAIP'07

September 17, 2007

Warsaw, Poland

Editors:
Emanuel Kitzelmann
Ute Schmid
Faculty of Information Systems and Applied Computer Sciences
University of Bamberg
Germany

Preface

The ECML/PKDD workshop AAIP 2007—"Approaches and Applications of Inductive Programming"—was held at the 18th European Conference on Machine Learning (ECML) and the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD) in Warsaw, Germany on 17 September, 2007. The main goal of AAIP was to bring together researchers working on different approaches to inductive programming as well as researchers from different areas of machine learning who are especially interested in the inductive synthesis of programs.

Inductive programming (or inductive program synthesis) incorporates all approaches which are concerned with learning programs or algorithms from incomplete specifications such as, e.g., input/output examples describing the desired behavior of the intended program or traces describing steps of computation of specific outputs. Typical additional input to an IP system is background knowledge such as predefined functions and predicates to be used or schemas/templates defining the general data flow of the intended program. Output of an IP system is a program in some arbitrary programming language containing conditionals and loop- or recursive control structures.

Inductive programming is a research topic of crucial interest for machine learning and artificial intelligence in general. The ability to generalize a program—containing control structures as recursion or loops—from examples is a challenging problem which calls for approaches going beyond the requirements of algorithms for concept learning. Pushing research forward in this area can give important insights in the nature and complexity of learning as well as enlarging the field of possible applications.

Typical application areas where learning of programs or recursive rules are called for, are first in the domain of software engineering where structural learning, software assistants and software agents can help to relieve programmers from routine tasks, give programming support for endusers, or support of novice programmers and programming tutor systems. Further areas of application are language learning, learning recursive control rules for AI-planning, learning recursive concepts in web-mining or for data-format transformations.

Today, different inductive programming algorithms are developed in and use techniques from different research fields, these are evolutionary computation, inductive logic programming, classical (Summers-like) inductive synthesis of functional programs, and grammar inference. They can be clustered into two major approaches: (i) the analytical/example-driven approach where programs are derived by analyzing given examples and (ii) the search-based/generate-and-test approach where programs are enumerated heuristically and then tested against given examples. Both approaches have different strengths and limits. The papers in these proceedings cover both approaches.

The AAIP 2007 workshop brought together researchers from different communities considering inductive programming from different perspectives and with the common interest on induction of general programs regarding theory, methodology and applications. The two invited speakers represent inductive program synthesis research in the

areas of evolutionary algorithm design (Roland Olsson, Østfold University College, Norway) and algebraic specification (Lutz Hamel, University of Rhode Island, USA).

We want to thank all those who contributed to AAIP 2007. We thank all members of the program committee for their support in promoting the workshop and for reviewing submitted papers and we thank the ECML organizers, especially the workshop chair Marzena Kryszkiewicz for technical support.

Bamberg, July 2007

Emanuel Kitzelmann
Ute Schmid

Workshop Organization

AAIP Workshop Chairs

Emanuel Kitzelmann (University of Bamberg, Germany)
Ute Schmid (University of Bamberg, Germany)

ECML/PKDD Workshop Chair

Marzena Kryszkiewicz (Warsaw University of Technology, Poland)

AAIP Workshop Program Committee

Ricardo Aler Mur (Universidad Carlos III de Madrid, Spain)
Pierre Flener (Sabanci University, Turkey, and Uppsala University, Sweden)
Emanuel Kitzelmann
Donato Malerba (Universita degli Studi di Bari, Italy)
Stephen Muggleton (Imperial College, UK)
Ute Schmid

Table of Contents

Automatic Design of Algorithms through Evolution (ADATE)

Roland Olsson

Faculty of Computer Science, Østfold University College, Norway
roland.olsson@hiof.no
http://www-ia.hiof.no/~rolando/

Automatic Design of Algorithms through Evolution (ADATE) is a system for automatic functional programming that is able to synthesize recursive programs with automatic invention of recursive help functions.

The invited talk will first present two recent applications of ADATE where it appears to be highly competitive with the best known alternative methods. The first application is synthesis of programs that drive an autonomous vehicle given input from gyros and range sensors. In the second application, ADATE generates algorithms for image segmentation, that is separating a possibly noisy image into regions representing objects of interest.

The autonomous driving example shows that ADATE is suitable for reinforcement learning and does not need explicitly provided outputs in order to generate desirable programs.

We will explain the basic program transformations employed by ADATE as well as briefly discuss the combinatorial search algorithms needed to efficiently and effectively search for suitable transformation combinations. The talk will also show the population management of ADATE and how it considers both the time complexity of synthesized programs and the need for syntactic complexity minimization in order to avoid overfitting.

An Inductive Programming Approach to Algebraic Specification

Lutz Hamel and Chi Shen

Department of Computer Science and Statistics
University of Rhode Island
Kingston, RI 02881, USA
hamel@cs.uri.edu, shenc@cs.uri.edu

Abstract. Inductive machine learning suggests an alternative approach to the algebraic specification of software systems: rather than using test cases to validate an existing specification we use the test cases to induce a specification. In the algebraic setting test cases are ground equations that represent specific aspects of the desired system behavior or, in the case of negative test cases, represent specific behavior that is to be excluded from the system. We call this inductive equational logic programming. We have developed an algebraic semantics for inductive equational logic programming where hypotheses are cones over specification diagrams. The induction of a hypothesis or specification can then be viewed as a search problem in the category of cones over a specific specification diagram for a cone that satisfies some pragmatic criteria such as being as general as possible. We have implemented such an induction system in the functional part of the Maude specification language using evolutionary computation as a search strategy.

1 Introduction

Inductive machine learning [1, 2] suggests an alternative approach to the algebraic specification of software systems: rather than using test cases to validate an existing specification we use the test cases to *induce* a specification. In the algebraic setting specifications are equational theories of a system where the test cases are ground equations that represent specific aspects of the desired system behavior or, in the case of negative test cases, represent specific behavior that is to be excluded from the system. Acceptable specifications must satisfy the positive test cases and must not satisfy the negative test cases. It is interesting to observe that in this alternative approach the burden of constructing a specification is placed on the machine. This leaves the system designer free to concentrate on the quality of the test cases for the desired system behavior. In addition to the positive and negative test cases an inductive equational logic program can also contain a background theory.

A simple example illustrates our notion of inductive equational logic programming. Here we are concerned with the induction of a stack specification from a set of positive test cases for the stack operations top, push, and pop. In

```
fmod STACK-PFACTS is
  sorts Stack Element .
  ops a b : -> Element .
  op v: -> Stack .
  op top : Stack -> Element .
  op pop : Stack -> Stack .
  op push : Stack Element -> Stack .

  eq top(push(v,a)) = a .
  eq top(push(push(v,a),b)) = b .
  eq top(push(push(v,b),a)) = a .
  eq pop(push(v,a))= v .
  eq pop(push(push(v,a),b)) = push(v,a) .
  eq pop(push(push(v,b),a)) = push(v,b) .
endfm
```

(a)

```
fmod STACK is
  sorts Stack Element .
  op top : Stack -> Element .
  op pop : Stack -> Stack .
  op push : Stack Element -> Stack .
  var S : Stack . var E : Element .

  eq top(push(S,E)) = E .
  eq pop(push(S,E)) = S .
endfm
```

(b)

Fig. 1. (a) Positive test cases for the inductive acquisition of the specification for the stack operations top, push, and pop. (b) An hypothesis that satisfies the test cases.

Figure 1(a) the positive facts are given as a theory in the syntax of the Maude specification language [3]. Here the function symbol push can be viewed as a stack constructor and each of the test cases gives an instance of the relationship between the constructor and the function top or pop. The set of negative examples and the background knowledge are empty. A hypothesis or specification that satisfies the positive facts is given in Figure 1(b). It is noteworthy that our implementation of an inductive equational logic system within the Maude specification system induces the above specification unassisted.

Since our system is implemented in an algebraic setting, that is, it is implemented in the functional part of the Maude specification languages, it made sense to develop an algebraic semantics for inductive equational logic programming. As we will develop later on in this paper, an inductive equational logic program can be viewed as a specification diagram in the category of equational theories. A hypothesis is a cone over a specification diagram and the induction of a hypothesis can then be viewed as a search problem in the category of cones over a specification diagram for a cone that satisfies pragmatic criteria such as being as general as possible without being trivial. As it turns out, the most general cone for a specification diagram is trivial (the empty theory). It is interesting to note that the simplest possible hypothesis which is obtained from "memorizing" all the facts is an initial object in the category of cones or a co-limit of the specification diagram. We believe that this view of inductive equational logic programming is novel and its algebraic nature crystallized many implementation issues for us in the Maude setting that were murky in the normal semantics [4] usually associated with inductive logic programming. This is especially true with dealing with negative facts in the algebraic setting.

The search strategy of our system is based on genetic programming employing evolutionary concepts to identify appropriate cones or hypotheses. Our system sets itself apart from other induction systems in that we consider multi-concept learning and robustness vital aspects for the usability of an induction system.

Multi-concept learning [5] allows the system to induce specifications for multiple function symbols at the same time (see Figure 1). Robustness enables the system to induce specifications even in the presence of inconsistencies in the facts [6].

This paper is structured as follows. Section 2 describes the algebraic semantics that underlies the design of our system. Due to space constraints we state all our results without proofs. A manuscript is in preparation which will elucidate our mathematical constructions in more detail. In Section 3 we sketch our implementation. We describe some experiments using our system in Section 4. In Section 5 we describe work closely related to ours. And finally, Section 6 concludes the paper with some final remarks and future research.

2 An Algebraic Semantics

Many sorted equational logic, at the foundation of algebraic specification, is the logic of substituting equals for equals with many sorted algebras as models and term rewriting as the operational semantics [7, 8]. Briefly, an equational theory or specification is a pair (Σ, E) where Σ is an equational signature and E is a set of Σ-equations. Each equation in E has the form $(\forall X)l = r$, where X is a set of variables distinct from the equational signature and $l, r \in T_\Sigma(X)$ are terms. [1] If $X = \emptyset$, that is, l and r contain no variables, then we say the equation is ground. When there is no confusion theories are denoted by their collection of equations, in this case E. We say that a theory E semantically entails an equation e, $E \models e$, iff $A \models e$ for all algebras A where $A \models E$. We say that a theory E deductively entails an equation e, $E \vdash e$, iff e can be derived from E via equational reasoning. Given two theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, then a theory morphism $\phi\colon T \to T'$ is a signature morphism $\phi\colon \Sigma \to \Sigma'$ such that $E' \models \phi(e)$, for all $e \in E$. Soundness and completeness for many-sorted equational logic is defined in the usual way [9]: $E \models e$ iff $E \vdash e$.

Inductive logic programming concerns itself with the induction of first-order theories or hypotheses from facts and background knowledge [4]. Although it is possible to induce theories from positive facts only, including negative facts helps to constrain the domain. Therefore, both positive as well as negative facts are typically given. This is also true for the case of inductive equational logic programming. Here the positive facts represent a theory that needs to hold in the hypothesis and the negative facts represent a theory that should not hold in the hypothesis. Before we develop our semantics we have to define what we mean by facts and background knowledge.

Definition 1. *A theory (Σ, F) is called a* **Σ-facts theory** *(or simply facts) if each $f \in F$ is a ground equation. A theory (Σ, B) is called a* **background theory** *if it defines auxiliary concepts that are appropriate for the domain to be learned. The equations in B do not necessarily have to be ground equations.*

[1] Here we only consider many-sorted, unconditional equations, but the material developed here easily extends to more complicated equational logics.

In the algebraic setting it is cumbersome to express theories in terms of a satisfaction relation that does not satisfy a set of equations. Therefore, we need a little bit more machinery in order to deal with negative facts more readily.

Definition 2. *Given a many-sorted signature Σ, then an equation of the form $(\forall\emptyset)t \neq t' = \texttt{true}$ is called an* **inequality constraint***, where $t, t' \in T_\Sigma(\emptyset)$ and $\{\neq, \texttt{true}\} \subset \Sigma$ with the usual boolean sort assignments and interpretations. A theory (Σ, E) is called an* **inequality constraints theory** *iff all equations in E are inequality constraints.*

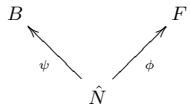
Our inequality constraints are not unlike Ehrig and Mahr's first order logical constraints [10]. We use inequality constraints to rewrite a negative Σ-facts theory as an inequality constraints theory. The idea being that we move from models that should not satisfy the negative facts to models that should satisfy the corresponding inequality constraints theory. We need the following proposition.

Proposition 1. *Given a theory (Σ, E) and an equation $(\forall\emptyset)l = r$, where $l, r \in T_\Sigma(\emptyset)$, such that $E \not\models (\forall\emptyset)l = r$, then $E \models (\forall\emptyset)(l \neq r) = \texttt{true}$ iff $E \not\models (\forall\emptyset)l = r$.*

Let E be some Σ-theory and let N be a Σ-facts theory such that $E \not\models e$, for all $e \in N$. We can now rewrite every equation $(\forall\emptyset)l = r$ in N as an inequality constraint $(\forall\emptyset)(l \neq r) = \texttt{true}$. Call this new set of equations $\hat{N}$, the inequality constraints theory. Observe that $E \models \hat{e}, \hat{e} \in \hat{N}$ iff $E \not\models e, e \in N$, as required.

A positive fact theory, a background theory, and an inequality constraint theory together make up an inductive equational logic program. This gives rise to the notion of a specification diagram.

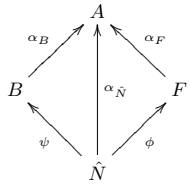
Definition 3. *Given a background theory B, (positive) facts F, and an inequality constraints theory $\hat{N}$ derived from negative facts N, we say that the following diagram is a* **specification diagram***,*



where ϕ and ψ are theory morphisms.

The intuition behind a specification diagram is that in an inductive equational logic program neither the background theory nor the positive facts should violate the inequality constraints. Now we define a cone over a specification diagram.

Definition 4. *Let $\psi\colon \hat{N} \to B$ and $\phi\colon \hat{N} \to F$ be a specification diagram, then a cone over the specification diagram is defined as,*



where α_B, α_F, and $\alpha_{\hat{N}}$ are theory morphisms and the diagram commutes. We call the apex or cone object A a **hypothesis**. *When there is no confusion we often denote cones by their apex objects.*

It is easy to see that the cones over a specification diagram S form a category, call it $\mathbf{H}(S)$, with cone morphisms between them; let P and Q be objects in $\mathbf{H}(S)$, then a cone morphism $c\colon Q \to P$ is a theory morphism such that $c|_S = id_S$. From an inductive programming point of view we are interested in the most general cone in $\mathbf{H}(S)$, where we define the relation *more general* as follows.

Definition 5. *Let P and Q be cones in $\mathbf{H}(S)$, then we say that P is* **more general** *than Q iff there exists a cone morphism $Q \to P$.*

Intuitively we might say that we are interested in the terminal object of the category $\mathbf{H}(S)$, since by definition this is the most general cone. Unfortunately, the terminal object in $\mathbf{H}(S)$ is a cone whose apex object is the empty theory. Thus, from a machine learning point of view this object is not very interesting. On the other hand, it is worthwhile to note that the initial object in $\mathbf{H}(S)$, that is the least general cone in $\mathbf{H}(S)$, is the co-limit of the specification diagram S and is easily constructed by simply pasting together or memorizing the theories in the specification diagram. Given this, it is easy to see that we have to resort to searching the category of cones over a specification diagram for an appropriate cone that is more general than the initial cone but not as general as the terminal cone. Therefore, our semantics seems to corroborate the well established notion of "generalization as search" [11].

Notions similar to the normal semantics developed for first order inductive logic programming [4] can be recovered from our semantics. Prior and posterior satisfiability as well as posterior sufficiency are direct consequences of our definition of a cone over a specification diagram. Prior necessity is a consequence of our definition of a specification diagram in that we do not admit morphisms from F to B.

3 Implementation

We have implemented an equational theory induction system within the functional part of the Maude specification language [3, 6, 12]. The mathematical view of inductive equational logic programming given in the previous section is more refined than those given in our previous accounts and reflects more accurately what happens in our implementation. The induction system is accessible from the Maude prompt via the `induce` command. The induce command returns an equational theory given a positive and a negative fact theory, as well as a background theory,

> `induce` *theory-name pfacts nfacts background parameters*

where *theory-name* is the name to be given to the induced theory, *pfacts* is the name of the positive fact theory, *nfacts* is the name of the negative facts theory, and *background* is the name of the background theory. Finally, *parameters*

denotes parameters that allow the user to assert some control over the induction process. In the terminology of the previous section the returned equational theory is the apex object of the most appropriate cone given the specification diagram derived from the *pfacts*, *nfacts*, and *background* theories.

Our induction process is an evolutionary search in the category of cones over a specification diagram for the most general cone whose apex object is not an empty theory (or an approximation to this cone, since evolutionary systems are not guaranteed to find the global optimum). More specifically, our system is based on genetic programming [13]. Genetic programming distinguishes itself from other evolutionary techniques in that it directly manipulates abstract syntax trees making it well suited for the induction of equational theories. In the following we refer to apex objects as hypotheses or pre-hypotheses (the meaning of which will be made precise below). It is clear that given a specification diagram and a hypothesis we can always recover the cone and given a cone we can always extract the hypothesis.

One key aspect of any search strategy and in particular evolutionary search strategies is that it needs to quantitatively distinguish between "good" and "bad" hypotheses. In order to accomplish this we endowed our induction system with the following objective function to be maximized:

$$\text{fitness}(H) = \text{facts}(H) + \text{constraints}(H) + \frac{1}{\text{length}(H)} + \frac{1}{\text{terms}(H)}\,, \tag{1}$$

where H denotes a (pre-) hypothesis, $\text{facts}(H)$ is the number of (positive) facts satisfied by H, $\text{constraints}(H)$ is the number of inequality constraints satisfied by H, $\text{length}(H)$ and $\text{terms}(H)$ denote the number of equations and terms in H, respectively. The fitness function is designed to primarily exert evolutionary pressure towards finding true hypotheses that satisfy all the facts and constraints (first and second terms). In addition, in the tradition of Occam's Razor, the fitness function also exerts pressure towards finding the shortest hypothesis (third and fourth terms). Note that we call a hypothesis a pre-hypothesis or pre-cone if it does not satisfy some of the facts or constraints.

Our search strategy based on genetic programming can be summarized as follows:

1. *Compute an initial (random) population of (pre-) hypotheses;*
2. *Evaluate the fitness of each (pre-) hypothesis;*
3. *Perform theory reproduction using genetic crossover and mutation operators;*
4. *Compute new population of (pre-) hypotheses;*
5. *Goto step 2 or stop if target criteria have been met.*

This series of steps does not significantly differ from the standard genetic programming paradigm [13]. The only real difference being that the fitness evaluation is mainly a proof obligation that the following theory morphism conditions hold: $H \models \alpha_F(f)$ for all $f \in F$ and $H \models \alpha_{\hat{N}}(n)$ for all $n \in \hat{N}$ given a hypothesis H, facts F, and inequality constraints $\hat{N}$. The morphism α_B is usually taken to be the theory inclusion and therefore there is no proof obligation. Soundness

and completeness of many-sorted equational logic allows us to replace semantic entailment with its proof-theoretic counterpart. This, in turn, allows us to automate the proofs by using the equations in the hypotheses as rewrite rules. It is interesting to note that hypotheses for which the theory morphism conditions do not hold will usually score a lower fitness value than hypotheses for which the theory morphism conditions do hold, especially in later generations of the evolutionary computation. From a genetic programming point of view it is important to not simply discard the theories for which the theory morphism conditions do not hold, because these pre-hypotheses could represent important partial solutions that upon later genetic recombination with other partial solutions could represent interesting hypotheses in their own right. In the evolutionary framework it is sufficient to simply label (pre-) hypotheses according to their fitness instead of discarding low performing ones outright.

Another important aspect of the evolutionary computation is the design of the genetic crossover and mutation operators. The design of these operators have a large impact on the quality of the solutions found by evolutionary computations. Our crossover operator allows for two types of crossovers:

1. **Expression-level crossover** - allows expression subtrees at the level of the left and right sides of equations to be exchanged between theories.
2. **Equation-level crossover** - allows the exchange of whole equations or sets of equations between theories.

The crossover operator works as expected with the only caveat that it has to respect typing information within the terms. Our system implements three different mutation operators:

1. **Expression-level mutation** - non-deterministally select an expression node in the abstract syntax of a theory, generate a new expression tree with the same sort, replace the original expression with the newly generated expression tree.
2. **Equation addition/deletion** - non-deterministically select an equation to be deleted from some theory, or generate a new equation and add it to some theory.
3. **Literal generalization** - non-deterministically choose a terminal expression node and replace it with a variable of the appropriate sort.

Again, the biggest difference between our mutation operator and the standard genetic programming mutation operator is that it has to respect the strict typing rules of many-sorted equational logic.

In our implementation we use the fitness convergence rate as a termination criterion. Should the fitness of the best individuals increase by less than 1% over 25 generations we terminate the evolutionary search since significant fitness improvement seems highly unlikely.

Our genetic programming engine is implemented as a strongly typed genetic programming system using Matthew Wall's GALib C++ library [14] within Maude. The system uses Maude's rewrite engine to dispense with the theory

morphism proof obligations during fitness evaluation. Since the equations in the hypotheses are generated at random, there is no guarantee that the theories do not contain circularities throwing the rewriting engine into an infinite rewriting loop while computing the fitness of a particular hypothesis. To guard against this situation we allow the user to set a parameter that limits the number of rewrites the engine is allowed to perform during the proof of each equation in the fact and constraints theories. This pragmatic approach proved very effective. The alternative would have been an in-depth analysis of the equations in each hypothesis adding significant overhead to the execution time of the evolutionary algorithm.[2]

As a final note on our implementation we need to acknowledge that premature convergence is a general problem in evolutionary computation. In this case, the population of an evolutionary algorithm converges on a suboptimal solution early on during the computation. Once this happens, there is little chance for the algorithm to discover other, more appropriate solutions. In order to prevent an evolutionary algorithm to converge prematurely a population is divided into multiple sub-populations (also called *demes* [15]) with only limited communication between them. The idea is that even if premature convergence occurs in some of the demes, diversity is maintained in the overall population due to the limited communication among the demes. The limited communication among the demes also serves to reseed diversity should some of the demes have prematurely converged. In our implementation we divide our population of hypotheses into ten demes where each deme carries a population of typically between 20 and 30 individual hypotheses.

4 Experiments

We have already mentioned that our system is able to induce the canonical stack theory given in the introduction, Figure 1. It is probably worthwhile to list some statistics in association with that experiment: We used an overall population of 200 individuals distributed over 10 demes; it took an average of 30 generations in the 50 trial runs to converge on the canonical solution; every single of the 50 trial runs converged on the canonical solution; each run took about 100 seconds on a 1.3GHz G4 Apple iBook.[3] It is also noteworthy that the hypothesis shown is virtually unedited with the exception for some renaming of variables for readability purposes. This is true with all hypotheses discussed here.

The stack induction problem looks straight forward from a conceptual point of view, however, from a machine learning point of view we are faced with a *multi-concept learning* problem in the sense that both the `top` and `pop` operations each represent a different concept to be acquired. That multi-concept learning is not

[2] At this point the authors are not even sure if circularity in a term rewriting system is a decidable property making an even stronger argument for our pragmatic approach.

[3] This experimental setup applies to all following experiments: a population of 200 individuals spread over 10 demes and 50 trial runs performed on a 1.3GHz G4 Apple iBook.

a guaranteed property of an induction algorithm is witnessed by the fact that other theory induction algorithms fail to produce a sensible theory in context of multi-concept learning (e.g. [16]).

```
fmod SUM-PFACTS is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op sum : Nat Nat -> Nat .

  eq sum(0,0) = 0 .
  eq sum(s(0),s(0)) = s(s(0)) .
  eq sum(0,s(0)) = s(0) .
  eq sum(s(s(0)),0) = s(s(0)) .
  eq sum(s(0),0) = s(0) .
  eq sum(s(0),s(s(0))) = s(s(s(0))) .
  eq sum(s(s(0)),s(s(0))) = s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(0)) = s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(s(0))) = s(s(s(s(s(0))))) .
endfm
```

(a)

```
fmod SUM-NFACTS is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op sum : Nat Nat -> Nat .

  eq sum(s(0),0) = 0 .
  eq sum(0,0) = s(0) .
  eq sum(s(0),s(0)) = s(0) .
  eq sum(s(0),s(0)) = 0 .
  eq sum(s(s(0)),s(s(0))) = s(s(0)) .
endfm
```

(b)

```
fmod SUM is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op sum : Nat Nat -> Nat .
  vars A B C : Nat .

  eq sum (A,0) = A .
  eq sum (A,s(C)) = sum(s(A),C) .
endfm
```

(c)

Fig. 2. Positive facts (a) and negative facts (b) for the induction of the `sum` function. A hypothesis for the `sum` function (c).

In our next experiment we illustrate that our system can acquire recursive specifications. In this experiment we induce the specification of the function `sum` that adds two natural numbers. The natural numbers are given in Peano notation, where the numbers are represented as $0 \mapsto 0$, $s(0) \mapsto 1$, $s(s(0)) \mapsto 2$, *etc.* The positive and negative facts are given by the theories in Figure 2 (a) and (b), respectively. The positive facts specify examples of applying the `sum` function to a number of small natural numbers. Also included are examples that show that summation is commutative. The negative facts consist of equations that should not hold in the induced specification for `sum`. Each equation in this theory is a counter example to the definition of the function `sum`. The background theory for this experiment is empty. Given the above theories our system will induce a hypothesis (or a variant that is isomorphic to this theory) as given in Figure 2(c). Some quick statistics: it took an average of 40 generations to produce a solution; we produced a minimal, recursive solution 32 times over 50 runs (for the other solutions the system noticed that it only had to produce a solution that specified the functionality of `sum` over the given small integers and it devised a non-recursive hypothesis); each run took about 120 seconds.

In our final experiment we demonstrate the usage of background knowledge during the induction process. The problem is to find a recursive way to sum the numbers in a list, given the knowledge of how to sum two numbers. Figure 3 displays the relevant theories. It is perhaps noteworthy that we use the theory induced in the previous experiment as background knowledge for the current experiment. Note that in Figure 3(d) the first two equations are due to the background information and the last two equations specify the actual solution. Some statistics on this experiment: it took an average of 35 generations to produce a solution; 38 of our 50 runs produced a solution similar to the one shown

```
fmod SUM-LIST-PFACTS is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op nl : -> NatList .
  op c : NatList Nat -> NatList .
  op suml : NatList -> Nat .

  eq suml(c(nl,0)) = 0 .
  eq suml(c(nl,s(0))) = s(0) .
  eq suml(c(nl,s(s(0)))) = s(s(0)) .
  eq suml(c(c(nl,0),s(0))) = s(0) .
  eq suml(c(c(nl,s(0)),s(0))) = s(s(0)) .
  eq suml(c(c(nl,s(s(0))),s(0))) = s(s(0)) .
  eq suml(c(c(nl,s(s(0))),s(s(0)))) = s(s(s(s(0)))) .
  eq suml(c(c(nl,0),s(s(0)))) = s(s(0)) .
  eq suml(c(c(nl,0),s(s(s(0))))) = s(s(s(0))) .
  eq suml(c(c(nl,s(s(0))),0)) = s(s(0)) .
endfm
```

(a)

```
fmod SUM-LIST-NFACTS is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op nl : -> NatList .
  op c : NatList Nat -> NatList .
  op suml : NatList -> Nat .

  eq suml(c(nl,0)) = s(0) .
  eq suml(c(nl,s(0))) = 0 .
  eq suml(c(nl,s(s(0)))) = s(0) .
  eq suml(c(c(nl,0),s(0))) = s(s(0)) .
  eq suml(c(c(nl,s(0)),s(0))) = s(s(s(0))) .
  eq suml(c(c(nl,s(0)),s(0))) = s(0) .
  eq suml(c(c(nl,s(0)),s(s(0)))) = s(s(0)) .
  eq suml(c(c(nl,0),s(s(0)))) = s(s(s(0))) .
  eq suml(c(c(c(nl,s(0)),s(0)),s(0))) = s(s(0)) .
  eq suml(c(c(c(nl,s(0)),0),s(0))) = s(0) .
endfm
```

(b)

```
fmod SUM-LIST-BACKGROUND is
  sort Nat .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op sum : Nat Nat -> Nat .
  vars A B : Nat .

  eq sum(0,A) = A .
  eq sum(s(A),B) = s(sum(A,B)) .
endfm
```

(c)

```
fmod SUM-LIST is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op sum : Nat Nat -> Nat .
  op nl : -> NatList .
  op c : NatList Nat -> NatList .
  op suml : NatList -> Nat .
  vars NatA NatB NatC : Nat .
  vars NatListA NatListB NatListC : Nat .

  eq sum(0,NatA) = NatA .
  eq sum(s(NatA),NatB) = s(sum(NatA,NatB)) .
  eq suml(nl) = 0 .
  eq suml(c(NatListA,NatB)) = sum(suml(NatListA),NatB) .
endfm
```

(d)

Fig. 3. Induction with background information: (a) positive facts, (b) negative facts, (c) background theory, and (d) resulting hypothesis.

in Figure 3(d) (the other solutions were non-recursive and did not generalize well beyond the test cases); each run took about 130 seconds.

These experiments highlight both the strength and weakness of the evolutionary approach to theory induction. The weakness is that in order to gain some confidence in an induced theory one needs to rerun the induction experiment multiple times. Only if the same or isomorphic theories are being discovered multiple times does one gain some confidence that the found theory constitutes a reasonable hypothesis. The strength of the evolutionary approach is that the likelihood of the search space being traversed in exactly the same way with every run is very low. Therefore, running the induction algorithm multiple times and inducing the same or isomorphic theories in different runs means that the induced (isomorphic) theories do represent a quasi global optimum. Perhaps a more statistical approach by applying leave-one-out cross-validation would be appropriate here in order to establish some confidence that the induced specifications generalize well. For additional and more complex examples please see Shen's thesis [12].[4]

[4] http://homepage.cs.uri.edu/faculty/hamel/dm/theses/Chi-thesis-2006.pdf

5 Related Work

The synthesis of equational and functional programs has a long history in computing extending back into the mid 1970's, e.g. [17–20]. The approaches use deductive as well as inductive techniques for the induction of recursive functional programs from formal specifications. This is in contrast to our machine learning setting where generalization is achieved by searching through an appropriate space. The advantages of the machine learning setting is that we can include positive and negative examples, as well as background information in a natural way. We also can incorporate "meta-properties" such as multi-concept learning and robustness [6]. For an insightful overview of the synthesis of equational programs see [21]. A survey that looks at the synthesis of predicate logic programs is [22].

The two approaches most related to ours are [16] and [23]. Both approaches use inductive learning with positive and negative examples of the functions to be induced. The former approach considers unsorted equational logic as the representation language using inverse narrowing as the search heuristic. Although this approach is very fast in inducing programs it is not robust and cannot be used in multi-concept settings. The latter approach uses a many-sorted, higher-order functional language as its representation language and uses an evolutionary algorithm as its induction heuristic. We should also mention Roland Olsson's inductive functional programming system Adate [24].

6 Conclusions and Further Work

We presented a system that given a set of positive and negative examples and relevant background knowledge will induce an algebraic specification. In this setting the examples are ground equations that can be considered test cases: the positive examples need to hold in the induced specification and the negative examples should not hold in the induced specification. We have implemented this system in the functional part of the Maude specification language. Our algebraic semantics for inductive equational logic programming elucidates many of the details necessary for the implementation of the system.

Future work will extend our approach to include full order-sorted, conditional equational logic. We will also investigate whether our approach can be extended to hidden-sorted equational logic. In this context it will be interesting to see how our evolutionary induction system can deal with function symbol invention (similar to predicate invention) which will most likely be necessary in order to evolve objects with hidden state and visible behavior. We would like to investigate the integration of our induction engine in Maude using its metalanguage facilities [25] .

References

1. Muggleton, S.: Inductive acquisition of expert knowledge. Addison-Wesley, Reading, Mass. (1990)

2. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
3. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada, J.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285**(2) (2002) 187–243
4. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19/20** (1994) 629–679
5. Michalski, R.S., Wnek, J.: Learning Hybrid Descriptions. In: Proceedings IIS, Augustow, Poland (June 1995)
6. Hamel, L., Shen, C.: Inductive acquisition of algebraic specifications, tech report tr06-317. Technical report, Dept. of Computer Science and Statistics, University of Rhode Island (2006)
7. Wechler, W.: Universal Algebra for Computer Scientists. Springer-Verlag (1992)
8. Burstall, R., Goguen, J.: Institutions: abstract model theory for specification and programming. JACM **39**(1) (1992) 95–146
9. Goguen, J., Meseguer, J.: Completeness of many-sorted equational logic. ACM SIGPLAN Notices **17**(1) (1982) 9–17
10. Ehrig, H., Mahr, B.: Fundamentals of algebraic specification 2: module specifications and constraints. Springer-Verlag New York, Inc. New York, NY, USA (1990)
11. Mitchell, T.M.: Generalization as search. Artificial Intelligence **18**(2) (1982) 203–226
12. Shen, C.: Inductive Equational Logic in Maude. Master's thesis, University of Rhode Island (2006)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, MA (1992)
14. Wall, M.: GAlib: A C++ Library of Genetic Algorithm Components. Mechanical Engineering Department, Massachusetts Institute of Technology, Aug (1996)
15. Goldberg, D.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1989)
16. Hernandez-Orallo, J., Ramrez, M.: Inverse Narrowing for the Induction of Functional Logic Programs. Proc. Joint Conference on Declarative Programming, APPIA–GULP–PRODE **98** (1998) 379–393
17. Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the Association for Computing Machinery **24**(1) (1977) 44–67
18. Summers, P.: A Methodology for LISP Program Construction from Examples. JACM **24**(1) (1977) 161–175
19. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. TOPLAS **2**(1) (1980) 90–121
20. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454
21. Dershowitz, N., Reddy, U.: Deductive and Inductive Synthesis of Equational Programs. JSC **15**(5/6) (1993) 467–494
22. Deville, Y., Lau, K.: Logic program synthesis. Journal of Logic Programming **19**(20) (1994) 321–350
23. Kennedy, C.J., Giraud-Carrier, C.: An evolutionary approach to concept learning with structured data. In: Proceedings of ICANNGA, Springer Verlag (1999) 1–6
24. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–58
25. Martı-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. Proceedings WRLA (2004) 391–414

Data-Driven Induction of Recursive Functions from Input/Output-Examples

Emanuel Kitzelmann

Faculty of Information Systems and Applied Computer Sciences,
University of Bamberg, 96045 Bamberg, Germany
emanuel.kitzelmann@wiai.uni-bamberg.de
http://www.cogsys.wiai.uni-bamberg.de/kitzelmann/

Abstract. We describe a technique for inducing recursive functional programs over algebraic datatypes from few non-recursive and only positive ground example-equations. Induction is data-driven and based on structural regularities between example terms. In our approach, functional programs are represented as constructor term rewriting systems containing recursive rewrite rules. In addition to the examples for the target functions, background knowledge functions that may be called by the induced functions can be given in form of ground equations. Our algorithm induces several dependent recursive target functions over arbitrary user-defined algebraic datatypes in one step and automatically introduces auxiliary subfunctions if needed. We have implemented a prototype of the described method and applied it to a number of problems.

1 Introduction

Automatic induction of recursive functional or logic programs from input/output-examples (I/O-examples) is an active area of research since the sixties (see [1] for classical methods, [2] for systems in the field of inductive logic programming, and [3] for recent research).

There exist two general approaches to tackle inductive synthesis of programs: (i) In the generate-and-test approach (e.g., the ADATE system [4]), programs of a defined class are enumerated heuristically and then tested against given examples. (ii) In the analytical approach, programs of a defined class are derived by detecting recurrences in given examples which are then generalized to recursively defined functions. That is, hypotheses[1] are (more or less) computed instead of searched. Generate-and-test methods are applicable for very general program classes since there are no principal difficulties in enumerating programs. They naturally facilitate usage of predefined functions (background knowledge) in induced programs. On the other side, generate-and-test methods are search intensive and therefore time consuming. Analytical approaches have more restricted program classes and generally do not facilitate the usage of background

[1] We adopt machine learning terminology here: The output program of an induction is called *hypothesis*, the function(s) to be induced are called *target function(s)*

knowledge. On the other side, analysis minimizes search and makes these approaches fast. The goal of the approach described in this paper was to relax the analytical approach by applying a search for hypotheses but by keeping analytical concepts within the search. The result is a functional program induction system that data-driven searches a comparatively less restricted hypothesis space and allows the use of background knowledge.

One classical and influential analytical approach is from Summers [5], who put inductive synthesis on a firm theoretical foundation. His system induces functional Lisp programs containing one function definition whose body consists of a conditional for an arbitrary number of base cases and one recursive case containing one recursive call. Parameters are restricted to be S-expressions (the general datatype in Lisp) and I/O-examples have to be linearly ordered. An interesting feature is a particular heuristic for automatically introducing an additional parameter if needed, e.g., the accumulator variable for list reversing. Analytical systems inspired by this approach are the BMWk algorithm [6,7] and the more recent system described in [8].

Another line of research is the field of inductive logic programming (ILP). Though ILP has a focus to non-recursive concept learning problems, there has also been research in inducing recursive logic programs on inductive datatypes (see [2]). One relatively recent analytical ILP method for inducing recursive logic programs is DIALOGS [9]. In order to induce more complex functions, e.g., the quicksort algorithm and to automatically invent needed subfunctions, e.g., the partitioning function for quicksort, it makes use of some general schemas, e.g., divide-and-conquer, which the user must chose and requires further information from the user. The schemas strongly restrict the hypothesis space. Moreover, DIALOGS is restricted to some predefined datatypes like lists and numbers.

The new approach described in this paper is a major extension of [10]. It induces multiple dependent target functions over arbitrary user defined algebraic datatypes in one step, facilitates the use of background knowledge and allows complex recursion patterns (nested calls of induced recursive functions, mutual recursion, tree recursion, arbitrary numbers of base- and recursive cases). Additionally needed subfunctions are introduced automatically if the calling relation fulfills some conditions. Its integrated analytical concepts lead to induction times which are very small compared to powerfull generate-and-test systems like ADATE. E.g., to induce the *Reverse*-function, our system needs milliseconds whereas ADATE needs more than a minute on the same computer. Particularly the capability of inducing multiple related target functions as for example mutual recursive definitions for *Even* and *Odd* on natural numbers is a feature not provided by most program induction systems. A recent ILP system also capable of learning multiple related recursively defined target concepts is ATRE [11].

2 Preliminaries

We represent functional programs as sets of equations (pairs of terms) over a many-sorted first order signature Σ. That is, we abstract from any concrete

functional programming language and do not consider higher order functions. Each equation specifying a function F has a left hand side (lhs) of the form $F(t_1, \ldots, t_n)$ where neither F nor the name of any other of the defined functions occur in the t_i. Thus, the symbols in the signature Σ are divided into two disjunct subsets $\mathcal{F}$ of *defined function symbols*, e.g., F, and $\mathcal{C}$ of *constructors*. Terms without defined function symbols are called constructor terms. Ground constructor terms denote values. The constructor terms t_i in the lhss of the equations for a defined function F may contain variables and are called *pattern*. This corresponds to the concept of pattern matching in functional programming languages and is the only form of case distinction. Each variable in the rhs of an equation must occur in the lhs, i.e., in the pattern. To evaluate a function defined by equations we read the equations as simplification rules from left to right. A set of simplification (or rewrite) rules is called *term rewriting system (TRS)*. TRSs whose lhss have defined function symbols as roots and constructor terms as arguments, i.e., whose lhss have the described pattern-matching form, are called *constructor term rewriting systems (CSs)*.

In order to formalize the simplification process we first introduce some standard concepts on terms: One can denote each subterm of a term t by its unique *position* within t, a sequence of positive integers. The term t itself stands at position ϵ—the empty sequence—called *root position*. If $t = f(t_1, \ldots, t_n)$ then each t_i stands at position i. If a subterm s of t_i stands at position u within t_i then it stands at position $i.u$ within t. The subterm at position u is written $t|_u$. Consider the term $t = f(a, g(x, y))$. Then, e.g., holds $t|_2 = g(x, y)$ and $t|_{2.1} = x$.

A *substitution* σ is a mapping from variables to terms and is extended to a mapping from terms to terms which is also denoted by σ and written in postfix notation; $t\sigma$ is the result of applying σ to all variables in term t. If $s = t\sigma$, then t is called *generalization* of s and we say that t *subsumes* s and that s *matches* t by σ. Given two terms t_1, t_2 and a substitution σ such that $t_1\sigma = t_2\sigma$, then we say that t_1, t_2 *unify* and call σ *unifier* of t_1 and t_2. We generalize the subsumption relation to sets of terms and say that a set of terms T subsumes another set of terms S if each term $s \in S$ is subsumed by a term $t \in T$. Given a set of terms, $S = \{s, s', s'', \ldots\}$, then there exists a term t which subsumes all terms in S and which is itself subsumed by each other term subsuming all terms in S. The term t is called *least general generalization (lgg)* of the terms in S [12].

A *context* is a term that contains a distinguished symbol $\Box$, denoting *wholes*, at at least one position. If C is a context containing n wholes then $C[t_1, \ldots, t_n]$ denotes the term resulting from replacing the n wholes in C by the t_i from left to right. The *rewrite relation* $\to_R$ established by a CS R is defined as follows: A term t rewrites to s according to R, written $t \to_R s$ iff there exists a rule $l \to r$ in R, a substitution σ, and a context C such that $t = C[l\sigma]$ and $s = C[r\sigma]$. Evaluating an n-ary function F for an input $i_1, \ldots, i_n$ consists of repeatedly rewriting the term $F(i_1, \ldots, i_n)$ w.r.t. the rewrite relation until the term is in *normal form*, i.e., cannot be further rewritten. A sequence of (in)finitely many rewrite steps $t_0 \to_R t_1 \to_R \cdots$ is called *derivation*. If a derivation starts with term t and results in a normal form s, then s is called normal form of t, written

$t \xrightarrow{!} s$. We say that t normalizes to s. In order to define a *function* on a domain (a set of ground terms) by a CS, no two derivations starting with the same ground term may lead to different normal forms, i.e., normal forms must be unique. A sufficient condition for this is that no two lhss of a CS unify; then the CS is *confluent*. A CS is *terminating* if each possible derivation terminates. A sufficient condition for termination is that the arguments/inputs of recursive calls strictly decrease within each derivation and w.r.t. a well founded order.

3 Analytical Induction of Recursive Functions

For better readability we write t^n for a sequence of terms $t_1, \ldots, t_n$.

If i^n is an input to a recursively defined function F with a corresponding output term o and $i^{n\prime}$ is the input to F resulting from a recursive call of F within computing i, then o contains the output term o' for $i^{n\prime}$ as subterm. Using this structural regularity between computations of recursively defined functions in order to infer the recursive definition from I/O-examples is the core of analytical function induction as proposed by Summers [5]. Examples for a target function F are equations of the form $F(i^n) = o$ where the i^n and o are ground constructor terms and are called example inputs and outputs respectively. A necessary condition for applying the described principle is that for each example input, all inputs resulting from recursive calls are also included in the example set. The following definition states this condition formally and extended to more than one target function.

Definition 1 (Recursively Subsumed Examples). *Let R be a CS which correctly computes a set of example equations. The example equations are called* recursively subsumed *w.r.t. R if for all example inputs i^n hold: Let $F(p^n) \to t$ be a rule in R such that i^n matches p^n by substitution σ. Then for each (recursive) call $F'(r^m)$ of a defined function F' of R in t the instantiation $r^m\sigma$ is contained as an example input in the example equations.*

4 Function Induction by Pattern Refinement, Matching, and Ubiquitous Subprogram Introduction

We require that induced CSs are terminating and that they represent functions, i.e., that they have unique normal forms. With regard to the given examples we require that a hypothesis is *correct*:

Definition 2. *A hypothesis, i.e., a CS R is* consistent/complete *w.r.t. a set of example equations iff for each example equation $F(i^n) = o$ holds*

consistent: $F(i^n) \xrightarrow{!}_R o$ *or* $F(i^n) \xrightarrow{!}_R s$ *for a* non-*constructor term s,*
complete: $F(i^n) \xrightarrow{!}_R s$ *for a constructor term s.*

A hypothesis is correct *iff it is both consistent and complete.*

The consistency condition assures that if the induced function is defined for an input then the corresponding function value is the specified output. The completeness condition assures that the induced function is total on the example inputs. We do not require the induced function to be total w.r.t. *all* Σ-constructor terms. Fig. 1 shows example equations for the *Reverse*-function and the equations induced by our system. Only the example equations and the corresponding datatype definitions were provided. Note that the example equations contain variables. Using variables where possible reduces the amount of needed example equations and makes the induction more time efficient. Two subfunctions have been introduced automatically, *Last* and *Init*, which compute the last element of a list and the list without the last element respectively. Note that automatically introduced subfunctions are simply named "Sub1", "Sub2" etc. by the system.

Example equations:

$$
\begin{array}{rrcl}
1. & \mathit{Reverse}([]) & = & [] \\
2. & \mathit{Reverse}([X]) & = & [X] \\
3. & \mathit{Reverse}([X,Y]) & = & [Y,X] \\
4. & \mathit{Reverse}([X,Y,Z]) & = & [Z,Y,X] \\
5. & \mathit{Reverse}([X,Y,Z,V]) & = & [V,Z,Y,X]
\end{array}
$$

Induced CS:

$$
\begin{array}{lcl}
\mathit{Reverse}([]) & \rightarrow & [] \\
\mathit{Reverse}([X|Xs]) & \rightarrow & [\mathit{Last}([X|Xs])|\mathit{Reverse}(\mathit{Init}([X|Xs]))] \\
\mathit{Last}([X]) & \rightarrow & X \\
\mathit{Last}([X_1,X_2|Xs]) & \rightarrow & \mathit{Last}([X_2|Xs]) \\
\mathit{Init}([X]) & \rightarrow & [] \\
\mathit{Init}([X_1,X_2|Xs]) & \rightarrow & [X_1|\mathit{Init}([X_2|Xs])]
\end{array}
$$

Fig. 1. Example equations and the induced solution for the *Reverse*-function

The induction of a terminating, confluent, correct CS is organized as a kind of best first search. During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating and confluent CS *but potentially with variables in the rhss not occurring in the lhss*. That is, the equations of a hypothesis during search do not necessarily represent *functions*. We call such equations and hypotheses containing them *unfinished* equations and hypotheses. A goal state is reached, if at least one of the best—according to a criterion explained below—hypotheses is finished, i.e., does not contain unfinished equations. Such a finished hypothesis is terminating and confluent by construction and since its equations entail the example equations, it is also correct.

Our induction bias is to prefer CSs whose patterns partition the example inputs into fewer subsets. This corresponds to preferring programs with fewer case distinctions. This leads, in some sense, to a most general hypothesis. Regarding

one defined function, this bias prefers a CS with fewer rules, since the pattern of each rule determines one unique subset. But consider the solution for the *Reverse*-function with the subfunctions *Last* and *Init* as shown in Fig. 1. The solution contains six rules but the number of induced example input subsets is only three, because *Last* and *Init* induce the same subsets (pattern $[X]$ subsumes the second example, pattern $[X_1, X_2|Xs]$ examples 3 - 5) which are again partitions of the subset induced by pattern $[X|Xs]$ of *Reverse* such that pattern [] from *Reverse* remains and induces the subset containing the first example input. So if we would have chosen fewer rules as preference bias then obviously the five example equations themselves would have been favored over the solution with *Init* and *Last* such that no generalization would have taken place.

With respect to the described bias and in order to get a complete hypothesis w.r.t. the examples, the initial hypothesis is a CS with one rule per target function such that its pattern subsumes all example inputs. In most cases (e.g., for all recursive functions) one rule is not enough and the rhss will remain unfinished. Then for one of the unfinished rules successors will be computed which leads to one or more (unfinished) hypotheses. Now repeatedly unfinished rules of currently best hypotheses are replaced until a currently best hypothesis is finished. Since one and the same rule may be member of different hypotheses, the successor rules originate successors of *all* hypotheses containing this rule. Hence, in each induction step several hypotheses are processed.

4.1 Initial Rules

Given a set of example equations for one target function, the initial rule is constructed by first antiunifying [12] all example inputs.[2] This leads to the lgg of the example inputs, i.e., to the most specific pattern subsuming all example inputs. Second, the example outputs are antiunified w.r.t. the substitutions resulting from antiunification of the inputs. This gives the lgg of all outputs were variables from the pattern are used if possible. Considering only lggs of example inputs as patterns narrows the search space. It does not constrain completeness of hypotheses regarding the example equations as shown by the following lemma.

Lemma 1. *Let R be a CS with non-unifying patterns and which is correct regarding a set of recursively subsumed example equations. Then there exists a CS R' such that R' contains exactly one pattern p' for each pattern p in R, each p' is the lgg of all example inputs matching the corresponding pattern p, and R and R' compute the same normal form for each example input.*

Proof. It suffices to show (i) that if pattern p of a rule r is replaced by the lgg of the example inputs matching p then also the rhs of r can be replaced by a new rhs such that the rewrite relation remains the same for the example inputs matching p, and (ii) that if the rhs of r contains a call to a defined function

[2] Note that for functions with arity > 1 inputs are sequences i^n, $n > 1$, of terms. We may consider such a sequence as *one* term by assuming a distinguished constructor symbol as root and the i^n as direct subterms.

then each instance of this call regarding the example inputs matching p is also an example input (matched by p or another pattern and, hence, also matched by patterns constituting lggs of example inputs). Proving these two conditions suffices because (i) assures equal rewrite relations for the example inputs and (ii) assures that each resulting term of one rewrite step which is not in normal form regarding R is also not in normal form regarding R'.

The second condition is assured if the example inputs are recursively subsumed. To show the first condition let p be a pattern in R which is *not* the lgg of the input examples matching it. Then there exists a position u with $p|_u = x, x \in Var(p)$ and $p'|_u = s \neq x$ if p' is the lgg of the input examples matching p. First assume that x does not occur in the rhs of the rule r with pattern p, then replacing x by s in p does not change the rewrite relation of r for the example inputs because still all example inputs are subsumed and are rewritten to the same term as before. Now assume that x occurs in the rhs of r. Then the rewrite relation of r for the input examples remains the same if x is replaced by s in p as well as in the rhs. □

4.2 Processing Unfinished Rules

This section describes the three methods for replacing unfinished rules by successor rules. All three methods are applied to a chosen unfinished rule. The first method, *splitting rules by pattern refinement*, replaces an unfinished rule with pattern p^n by at least two new rules with more specific patterns in order to establish a case distinction on the example inputs. The second method, *introducing function calls*, implements the principle described in Sec. 3 in order to introduce recursive calls or calls to other defined functions. Other defined functions can be further target functions or background knowledge functions. The third method, *introducing subfunctions*, generates new induction problems, i.e., new example equations, for the unfinished subterms of an unfinished rhs. These new problems are treated the same way as the "original" problems, i.e., this method implements the capability to automatically find auxiliary subfunctions.

Splitting Rules by Pattern Refinement The first method for generating successors of a rule is to replace its pattern p^n by a set of more specific patterns, such that the new patterns induce a partition of the example inputs matching p^n. This results in a *set* of new rules replacing the original rule and—from a programming point of view—establishes a case distinction.

Suppose a rule with pattern p^n which is the lgg of the example inputs matching it. Then the examples whose inputs match p^n have to be partitioned into a minimum number of at least two subsets and p^n has to be replaced by the lggs of the inputs of the respective subsets. It has to be assured that no two of the new lggs unify. This is done as follows: First a position u is chosen at which a variable stands in p^n. Since p^n is the lgg of all inputs matching it it holds that at least two inputs have different constructor symbols at position u. Then respectively all example inputs with the *same* constructor at position u are taken into the

same subset. This leads to a partition of the example inputs. Finally, for each subset the lgg is computed. The new lggs do not unify, since they have different constructors at at least one position.

Possibly different positions of variables in pattern p^n lead to different partitions. Then all partitions and the corresponding sets of specialized patterns are generated. Each new pattern determines the lhs of a new rule. The corresponding initial rhss are computed as lggs of the respective outputs as described in Sect. 4.1. Since the refined patterns subsume fewer examples, the number of variables in the initial rhss which are not contained in the corresponding lhs (non-strictly) decreases with each refinement step. Eventually, if no correct hypothesis with fewer partitions exists, each example input is subsumed by itself such that the example equations are simply reproduced.

For example, let

$$1.\ \mathit{Reverse}([]) = [], \quad 2.\ \mathit{Reverse}([a]) = [a], \quad 3.\ \mathit{Reverse}([b]) = [b],$$
$$4.\ \mathit{Reverse}([a,b]) = [b,a], \quad 5.\ \mathit{Reverse}([b,a]) = [a,b]$$

be some examples for the *Reverse*-function. The pattern of the initial rule is simply a variable X, since the example input terms have no common root symbol. Hence, the unique position at which the pattern contains a variable and the example inputs different constructors is the root position. The first example input consists of only the constant [] at the root position. All remaining example inputs have the constructor *cons* as root. I.e., two subsets are induced by the root position, one containing the first example, the other containing all remaining examples. The lggs of the example inputs of these two subsets are [] and $[X|Xs]$ respectively which are the patterns of the two successor rules.

Introducing Function Calls The second method to generate successor sets for an unfinished rule with pattern p^n for a target function F is to replace its rhs by a call to a defined function F', i.e. by a term $F'(R_1(p^n), \ldots, R_m(p^n))$. Each R_i denotes a new introduced defined (sub)function. This finishes the rule, since now the rhs does not longer contain variables not contained in the lhs. In order to get a rule leading to a *correct* hypothesis, for each example equation $F(i^n) = o$ of function F whose input i^n matches p^n with substitution σ must hold: $F'(R_1(p^n), \ldots, R_m(p^n))\sigma \xrightarrow{!} o$. This holds if for each output o an example equation $F'(i'_1, \ldots, i'_m) = o$ of function F' exists such that $R_i(p^n)\sigma \xrightarrow{!} i'_i$ for each R_i and i'_i. Thus, if we find example equations of F' with outputs o, then we abduce example equations $R_i(i^n) = i'_i$ for the new subfunctions R_i and induce them from these examples. Provided, the final hypothesis is correct for F' and all R_i then it is also correct for F.

In order to assure termination of the final hypothesis it must hold $i^{m\prime} < i^n$ according to any reduction order $<$ if the function call is recursive.[3]

[3] Assuring decreasing arguments is more complex if mutual recursion is allowed.

Introducing Subfunctions The last method to generate successor equations can be applied, if all outputs o of the inputs matching the pattern of the considered unfinished rule have the same constructor c as roots. Let c be of arity m then the rhs of the rule is replaced by the term $c(Sub_1(p^n), \ldots, Sub_m(p^n))$ where each Sub_i denotes a new introduced defined (sub)function. This finishes the rule since all variables from the new rhs are contained in the lhs. The examples for the new subfunctions are abduced from the examples of the current function as follows: If $o|_i$ are the ith subterms of the outputs o, then the equations $Sub_i(i^n) = o|_i$ are the example equations of the new subfunction Sub_i. Thus, correct rules for Sub_i compute the ith subterm of the outputs o such that the term $c(Sub_1(p^n), \ldots, Sub_m(p^n))$ normalizes to the outputs o.

A Remark on the Described Successor Functions As described in Sect. 4.2, (recursive) calls to defined functions specified by examples are only introduced at the root of a rhs (since such calls are introduced by replacing an unfinished rhs). Of course, generally, function calls can occur at any position in a rhs, compare for example the recursive definition for *Init* in Fig. 1. The reason why e.g. *Init* can be induced by our approach though function calls are only introduced at root positions is that deeper positions are (indirectly) considered as consequence of subprogram introduction as described in Sect. 4.2. Rhs root positions of such subprograms correspond to deeper positions of the rhs of the rule calling these subprograms.

5 Experimental Results

We have implemented a prototype of the described algorithm in the programming language Maude [13]. Maude is a reflective language which is based on equational and rewriting logic. Reflection means that Maude programs can deal with Maude programs as data. The implementation includes two extensions compared to the approach described in the previous section: First, example equations may contain variables such that the amount of example equations needed to specify a target function decreases. This is advantageous for the specifier as well as it leads to smaller induction times. Second, different variables within a pattern can be tested for equality This establishes—besides pattern refinement—a second form of case distinction.

In Tab. 1 we have listed experimental results for sample problems. The first column lists the names for the induced target functions, the second the names of additionally specified background functions, the third the number of given examples (for target functions), the fourth the number of automatically introduced recursive subfunctions, the fifth the maximal number of calls to defined functions within one rule, and the sixth the times in seconds consumed by the induction. Note that the example equations contain variables if possible (except for the *Add*-function); compare Fig. 1. The experiments were performed on a Pentium 4 with Linux and the Maude 2.3 interpreter.

target functions	bk funs	#expl	#subfuns	#funcalls	times
Length	/	3	0	1	.012
Last	/	3	0	1	.012
Odd	/	4	0	1	.012
ShiftL	/	4	0	1	.024
Reverse	*Snoc*	4	0	2	.024
Even, Odd	/	3, 3	0	1	.028
Mirror	/	4	0	2	.036
Take	/	6	0	1	.076
ShiftR	/	4	2	2	.092
DelZeros	/	7	0	1	.160
Insertion Sort	*Insert*	5	0	2	.160
PlayTennis	/	14	0	0	.260
Add	/	9	0	1	.264
Member	/	13	0	1	.523
Reverse	/	4	2	3	.790
Quick Sort	*Append*,P_1,P_2	6	0	5	63.271

Table 1. Some inferred functions

All induced programs compute the intended functions with more or less "natural" definitions. *Length*, *Last*, *Reverse*, and *Member* are the well known functions on lists. *Reverse* has been specified twice, first with *Snoc* as background knowledge which inserts an element at the end of a list and second without background knowledge. The second case (see Fig. 1 for given data and computed solution), is an example for the capability of automatic subfunction introduction and nested calls of defined functions. *Odd* is a predicate and true for odd natural numbers, false otherwise. The solution contains two base cases (one for 0, one for 1) and in the recursive case, the number is reduced by 2. In the case where both *Even* and *Odd* are specified as target functions, both functions of the solution contain one base case for 0 and a call to the other function reduced by 1 as the recursive case. I.e. the solution contains a mutual recursion. *ShiftL* shifts a list one position to the left and the first element becomes the last element of the result list, *ShiftR* does the opposite, i.e., shifts a list to the right such that the last element becomes the first one. The induced solution for *ShiftL* contains only the *ShiftL*-function itself and simply "bubbles" the first element position by position through the list, whereas the solution for *ShiftR* contains two automatically introduced subfunctions, namely again *Last* and *Init*, and conses the last element to the input list without the last element. *Mirror* mirrors a binary tree. *Take* keeps the first n elements of a list and "deletes" the remaining elements. This is an example for a function with two parameters where both parameters are reduced within the recursive call. *DelZeros* deletes all zeros from a list of natural numbers. The solution contains two recursive equations. One for the case that the first element is a zero, the second one for all other cases. *Insertion Sort* and *Quick Sort* respectively are the well known sort algorithms. The five respectively six well chosen examples as well as the additional exam-

ples to specify the background functions are the absolute minimum to generate correct solutions. The solution for *Insertion Sort* has been generated within a time that is not (much) higher as for the other problems, but when we gave a few more examples, the time to generate a solution explodes. The reason is, that all outputs of lists of the same length are equal such that many possibilities of matching the outputs in order to find recursive calls exist. The number of possible matches increases exponentially with more examples. The comparatively very high induction time for *Quick Sort* results from the many examples needed to specify the background functions and from the complex calling relation between the target function and the background functions. P_1 and P_2 are the functions computing the lists of smaller numbers and greater numbers respectively compared to the first element in the input list. For *Add* we have a similar problem. First of all, we have specified *Add* by ground equations such that more examples were needed as for a non-ground specification. Also for *Add* holds, that there are systematically equal outputs, since, e.g., $Add(2,2), Add(1,3)$ etc. are equal and due to commutativity. Finally, *PlayTennis* is an attribute vector concept learning example from Mitchell's machine learning text book [14]. The 14 training instances consist of four attributes. The five non-recursive rules learned by our approach are equivalent with the decision tree learned by ID3 which is shown on page 53 in the book. This is an example for the fact, that learning decision trees is a subproblem of inducing functional programs.

6 Conclusions and Further Research

We described a method to induce functional programs represented as confluent and terminating constructor systems. The presented methodology is inspired by classical and recent analytical approaches to the fast induction of functional programs. One goal was to overcome the drawback that "pure" analytical approaches does not facilitate the use of background knowledge and generally have relatively restricted hypothesis languages and on the other side to keep the analytical approach as far as possible in order to be able to induce more complex functions in a reasonable amount of time. This has been done by applying a search in a more comprehensive hypothesis space but where the successor functions are data-driven and not generate-and-test based, such that the number of successors is more restricted and the hypothesis spaced is searched in a controlled manner. Though the successor functions are data-driven, the search is complete and only favors hypotheses inducing fewer partitions but applies no further heuristics to estimate, how many partitions the final hypothesis will have. Developing such heuristics will be one of the further research topics.

References

1. Biermann, A.W., Guiho, G., Kodratoff, Y., eds.: Automatic Program Construction Techniques. Collier Macmillan (1984)

2. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. Journal of Logic Programming **41**(2–3) (1999) 141–195
3. Kitzelmann, E., Olsson, R., Schmid, U., eds.: Proceedings of the ICML 2005 Workshop Approaches and Applications of Inductive Programming. (2005)
4. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–83
5. Summers, P.D.: A methodology for LISP program construction from examples. Journal ACM **24**(1) (1977) 162–175
6. Kodratoff, Y., J.Fargues: A sane algorithm for the synthesis of LISP functions from example problems: The Boyer and Moore algorithm. In: Proc. AISE Meeting Hambourg. (1978) 169–175
7. Jouannaud, J.P., Kodratoff, Y.: Characterization of a class of functions synthesized from examples by a summers like method using a 'B.M.W.' matching technique. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI-79), Morgan Kaufmann (1979) 440–447
8. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454 Special topic on Approaches and Applications of Inductive Programming.
9. Flener, P.: Inductive logic program synthesis with DIALOGS. In Muggleton, S., ed.: Proceedings of ILP'96, Springer (1997) 175–198
10. Kitzelmann, E., Schmid, U.: Inducing constructor systems from example-terms by detecting syntactical regularities. Electronical Notes in Theoretical Computer Science **174**(1) (2007) 49–63
11. Malerba, D.: Learning recursive theories in the normal ILP setting. Fundamenta Informaticae **57**(1) (2003) 39–77
12. Plotkin, G.D.: A note on inductive generalization. In: Machine Intelligence. Volume 5. Edinburgh University Press (1969) 153–163
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In Nieuwenhuis, R., ed.: Rewriting Techniques and Applications (RTA 2003). Number 2706 in Lecture Notes in Computer Science, Springer-Verlag (June 2003) 76–87
14. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education (1997)

A Functional Approach to Evolving Recursive Programs

Martin Dostál

Department of Computer Science, Palacky University, Olomouc, Czech Republic
dostal@inf.upol.cz

Abstract. This paper introduces a multi objective function based approach for evolving programs with emphasis on modularity and repetitive code execution. A functional paradigm based programming language for evolution of programs is presented. The evolution of recursive functions using code structure abstraction is discussed henceforward and a kind of structure abstracting functions (AR-functions) is introduced. Some examples and results are presented.

1 Introduction

One of the most important research areas in the field of automatic programming is evolving of modular programs. The support for modularity is among others essential to enable more efficient evolution of complex programs containing iteration or recursion.

There are several approaches to repetitive code execution in genetic programming, the most widely used automatic programming approach, like iteration functions [1, 2] and recursion [2, 3]. Another interesting approach to evolution of recursive programs represent the abstraction of code structure of common types of recursion - the so called implicit recursion. Some research with genetic programming and predefined implicit recursion functions in a function set was done by Yu [4]. There are several other approaches to automatic programming, for instance PIPE [5] or ADATE [6] which well supports the induction of recursive programs using exhaustive search and heuristic transformation rules.

In this paper we present a very simple approach to efficient evolution of modular programs. We will utilise the advantages of modularity and functional programming language described thereunder to enable more efficient evolution of recursive programs using abstraction of code structure. The central idea behind our approach is to take advantages of the interconnection of multiple objective function approach and adaptive representation of code. The multiple objective function, nor adaptive representation of code are not generally new ideas, however we present a slightly different approach. The main differences are shortly summarised in Table 1. A handy paradigm to deal with modularity is offered by functional programming. We have developed a special functional automatic programming language called apFSM. The syntax and semantics of apFSM are inspired by Common LISP [7] and Scheme [8] functional programming languages. The crucial apFSM features are following:

property	ADF	MA	ARL	our approach
random module identification	yes	yes	no	no
fitness function	single	single	single	multiple
measuring the quality of individual	discrete	discrete	discrete	bool
subject of evolution (I-individual, L-language)	I, L	I, L	I, L	L
predefined structure of modules	yes	no	no	yes

Table 1. Comparison of approaches to modularity

- universal computing power (Turing-completeness)
- uniform data and code representation
- small function set and minimal number of data types
- in conjunction with a modular approach to objective function it serves as a language with evolvable code representation
- in conjunction with a modular approach to objective function it produces a modular solution of problems
- a priori support for repetitive execution of code using recursion
- modularity via first-order functions (bottom-up modularity)
- a priori support for code abstraction (top-down modularity) via higher-order functions with support of lexical scoping by lexical closures[1]

2 apFSM: A Functional Paradigm Based Computational Model for Automatic Programming

The apFSM language has simple syntax, built-in support for lexical scoping and NIL-bound variables[2]. Programs as well as data are represented as lists.

The basic language elements are *data types* (atom, cons, lexical closure), built-in (the so called *primitive*) functions (atom, eq, cons, car, cdr) and *special operators* (quote, if, lambda, label). Atoms name variables, functions and special operators. Functions may be primitive or user defined by the special operator lambda. Program code execution is application of functions on data. Program execution is driven by the *evaluation process* and it is called *evaluation*.

2.1 Data Types

Atom is a sequence of letters and digits. Atoms `T` and `NIL` have special meaning (see later). For instance, `A`, `25`, `LISP`, `A3324ABC` are atoms.

Primitive functions related to atoms:

atom is a function of one argument, returns `T` if its argument is an atom, otherwise returns `NIL`.

[1] it avoids the variable capture problem

[2] unbound variables (atoms) are evaluated to NIL instead of signaling error

eq is a function of two arguments, returns `T` if arguments are identical atoms, otherwise returns `NIL`.

Cons is a pair of arbitrary objects, first component is called *car*, second *cdr*. Cons of objects α and β is notated as $(\alpha\ .\ \beta)$. If cdr of a cons is another cons then it can be used a *simplified notation* [3], where dot and parentheses of the inner cons are omitted. If cdr of a cons is `NIL` then dot and atom `NIL` is omitted.

Primitive functions related to conses:

cons is a function of two arguments. It creates a new cons where first argument is car and second argument is cdr of new cons.

car is a function of one argument. If argument is cons, returns car of cons argument, otherwise returns `NIL`.

cdr is a function of one argument. If argument is cons, returns cdr of cons argument, otherwise returns `NIL`.

List of elements $a_1, a_2 \ldots a_n$ is defined as follows:

1. for $n = 0$ it is the atom `NIL`[4].
2. for $n > 0$ it is a cons whose car is a_1 and cdr is a list of elements $a_2 \ldots a_n$.

Lexical closure is an object with special role in evaluation process. It will be described in the evaluation process description. Instead of term lexical closure it can be used term *function*.

Truth values *false* represents atom `NIL`, other objects are *true*. Primitive functions use atom `T` to express true. For instance, expressions `T`, `(A B C)`, `LISP` are true.

2.2 Evaluation Process

Expressions (of the apFSM language) are atoms and lists composed of expressions. Expressions are *evaluated* by the evaluation process. If an expression e yields value v we say that e returns v, or that value of e is v.

Binding is a pair of an atom and a value which is an object. It is called the value of atom.

Lexical environment is a set of bindings in which every atom appears at most once.

Lambda expression is a list `(lambda` *args exp*`)` where *args* is a list of atoms called *argument list*, *exp* is an expression.

Lexical closure is a pair of a lambda expression and a lexical environment.

Global lexical environment is lexical environment where exist bindings of atoms `atom`, `eq`, `cons`, `car` and `cdr`, to primitive functions[5], atom `t` to `t` eventually bindings of atoms to objects. Other atoms are bound to `NIL`.

Special operators are atoms `quote`, `if`, `lambda` and `label`. These atoms have special evaluation rules.

[3] for instance, cons `(A . (B . (C . NIL)))` in simplified notation: `(A B C)`

[4] it is called the *empty list* and it can be alternatively notated as ().

[5] car to primitive function (pf.) car, cdr to pf. cdr, cons to pf. cons, atom to pf. atom eq to pf. eq

Expressions are evaluated in a lexical environment. If an expression has not specified a lexical environment, then it is the global lexical environment used. Evaluation process of an expression *exp* in lexical environment *env* is as follows:

1. if *exp* is an atom then the value of *exp* is the value of binding of the atom *exp* in the lexical environment *env*.
2. if *exp* is a list whose car is a special operator then evaluation of *exp* is driven by special evaluation rules. Each special operator has defined count, type and eventually the structure of arguments. If this requirement is not fulfilled, then value of *exp* is undefined.
   (a) **quote:** expression `(quote` *a*`)` returns (unevaluated) *a*.
   (b) **if:** expression `(if` *pred exp*1 *exp*2`)` is evaluated as follows: first of all *pred* is evaluated. If *pred* is true *exp*1 is evaluated, otherwise *exp*2.
   (c) **label:** expression `(label` *name lambda_exp* $v_1 \ldots v_n$`)` is evaluated as follows: If *name* is an atom and *lambda_exp* a lambda expression, it is evaluated *lambda_exp* in environment *env*. Then `(`*name* $v_1 \ldots v_n$`)` is evaluated in environment *env* with new bindings of arguments of *lambda_exp* to values $v_1 \ldots v_n$ and binding of atom *name* to value of *lambda_exp*.
   (d) **lambda:** expression *exp* must be a lambda expression. Value of *exp* is a lexical closure whose lambda expression is *exp* and lexical environment is *env*.
3. otherwise all elements of *exp* are evaluated. Denote evaluated elements of *exp* as $a_1 \ldots a_n$. If value of a_1 is a:
   (a) **primitive function:** evaluation of *exp* is the value of primitive function a_1 applied to $a_2 \ldots a_n$.
   (b) **lexical closure:** it is created a new lexical environment with bindings of *exp* and bindings of arguments of lambda expression to values $a_2 \ldots a_n$. In this new lexical environment lambda expression of closure *exp* is evaluated.

3 Automatic Programming Technique

The proposed automatic programming technique is based on the utilisation of multiple objective function simultaneously with principle of evolution of language for representing programs. The central idea behind our technique is the evolution of language for representing programs to be more appropriate to express a solution of a problem.

One of the main design intentions is to propose as simple as possible technique which will well demonstrate the advantages of proposed multi objective function approach. Nevertheless, it is possible to incorporate our multi objective function approach to another automatic programming technique, for instance genetic programming. The multiple objective function is defined as (one) main objective function and arbitrary number of subordinate objective functions, the so called *sub-objective functions*. Sub-objective functions represent such subordinate problems of a problem which solution may be useful in evolving solution

of the main objective function which will reduce search effort required to find a solution of a problem. In other words, sub-objective functions represent an additional heuristic knowledge about structure of a problem domain (or a presumable modularisation of a problem domain). Evolutionary algorithms deal with the discovery of building blocks. The (useful) building blocks in the point of view of automatic programming can be understood as modules of problem which acquisition may be very useful to evolve solution of a problem. Our modularity approach enables more effective acquisition of building blocks. When a solution of a sub-objective function found then it is immediately used to evolve function set of the representational language. More clearly, the proposed automatic programming technique can be described by following algorithm:

1. If maximum count of programs evolved then go to step 8.
2. Generate new program p.
3. If p is a solution of main objective function then return p else continue.
4. Set current objective function to first sub-objective function.
5. If p is a solution of the current objective function then go to step 6 else go to step 7.
6. Add new function to language. It represents a solution of current objective function. Set the name of new function identical to the name of sub-objective function.
7. If all sub-objective functions were tested then go to step 1 else set current objective function to next sub-objective function and go to step 5.
8. End.

The multiple objective function is defined by the user and therefore its definition drastically affects efficiency of program evolution. Some of the approaches to modularity attempt to discover sub-routines automatically, nevertheless we resign from this idea. In particular, the main reason is that the methods for automatic identification of proper building blocks (in other words, the central question is "which subroutine is better to take place as a module of problem solution than another ?") are not reliable enough to recognise valid building blocks. Many of automatically discovered building blocks by present approaches often represent a disingenuous pieces of code. Such building blocks are mostly useless. Let us conclude the main differences between the proposed approach and traditional genetic programming.

1. Multi objective fitness function (multi objective function based approaches to genetic programming are also known, but they diverge from our approach, see introduction and Table 1) in comparison to the traditional approach to objective (fitness) function.
2. The adaptation of language for representing programs instead of adaptation of individuals in population. Nevertheless both approaches could be combined together.
3. No recombination operators (no crossover or mutation). Our programs are randomly generated and tested for multiobjective function and then retained in evolved language or discarded.

4. Multi objective function does not measure a quality of programs as "closeness of a program to a solution" as it does genetic programming. In our approach, objective functions measure if program represents a solution of main objective function or a sub-objective function, or neither. Main objective and Sub-objective function measures a program using a set of test cases. Test case is a pair of input and output value. A program represents a solution of a (sub- or main) objective function if satisfyies every test case of the objective function.

Our approach enables two types of modularity:

Bottom-up modularity is realised using first-order functions. The solution of a sub-objective function (a building block) is represented as a function. A simple example follows:

Example: Even-3-Parity Problem let us consider a non-modular case at the beginning. The main objective function was `E-PARITY3`, none sub-objective functions were defined. The sample solution depicted below follows that it is not well modular:

```
(LABEL E-PARITY3
 (LAMBDA (A B C)
   (IF (IF C
           (IF (IF T (IF B NIL T) B) (CDR NIL) T)
         (IF T (IF B NIL T) B))
       (IF A NIL T) A)))
```

Now we will attempt to take advantage of proposed multi objective function approach. We put the main objective function to `E-PARITY3` and the set of sub-objective functions to `E-NOT` (negation) and `E-XOR` (exclusive `OR`). Following code represents a sample solution of the main objective (`E-PARITY3`) and sub-objective functions (`E-NOT`, `E-XOR`):

```
(LABEL E-NOT (LAMBDA (A) (IF A NIL T)))
(LABEL E-XOR (LAMBDA (A B) (IF A (E-NOT B) B)))
(LABEL E-PARITY3 (LAMBDA (A B C) (E-XOR (E-XOR C (E-XOR T B)) A)))
```

In comparison to first `E-PARITY3` solution it can be seen that the second solution is much more straightforward and modular than first `E-PARITY3` solution. Sub-objective functions can effectively modularise solution of a problem and rapidly reduce required search effort, see [9] for results and further discussion.

Top-down modularity is realised using higher-order functions for code structure abstraction. Certain types of functions, such as some types of recursion may be defined using structure abstraction. It may be useful to improve evolution of complex functions by abstracting useful code structure of a function into higher-order function that is programable by another functions.

The structure abstraction approach realises a top-down modularity approach: a general (structure abstraction) function is used as a sub-objective function and when its solution is found we have obtained (abstracted) a presumable code structure useful for evolving solution of objective function.

4 Evolving of Recursive Functions

The apFSM language introduced thereunder has built-in support for recursive functions. Nevertheless, the above mentioned support for code structure abstraction may be very useful to enable more efficient evolution of recursive functions than without structure abstraction.

It can be seen that code structure of many recursive functions falls into a few repeating standard forms. If we inspect a recursive function, it can be seen that it is composed of a *terminating condition(s)* and a *body function* which is recursively applied to the *current element* and *successive elements*. The critical part of evolving recursive function is to obtain a proper structure of recursion, especially a correct definition of recursive call. See Table 3 for comparison of results. Therefore for reduction of search effort it would be helpful to abstract *structure* of common types of recursive functions. The recursion abstracting functions may serve as highly parametrisable recursive functions. The structure abstraction can be thought as sub-objective functions in the same way as any other objective function in our approach.

4.1 Implicit Recursion

The concept of structure abstraction is not generally new. The recursion structure abstracting functions can be found in many functional programming languages like Haskell [10], Scheme [8] or Common LISP [7] and they are titled *implicit recursion functions* (IR-functions). The best known ones are `FOLDL`, `FOLDR` and `MAP`.

FOLDR - left folding recursion

```
(+ 10 '(1 2 3)) => (+ 1 (+ 2 (+ 10 3)))
```

FOLDL - right folding recusrion

```
(+ 10 '(1 2 3)) => (+ (+ (+ 10 1) 2) 3)
```

MAP - consing implicit recursion

```
((+ x 1) '(1 2 3)) => ((+ 1 1) (+ 1 2) (+ 1 3)) = (2 3 4)
```

4.2 AR-functions

IR-functions may be useful for some types of recursive functions, nevertheless these functions are not general enough (see Sect. 4.3 for example). Now we introduce more parametrisable structure abstraction functions which are better suited for automatic programming, the so called AR-functions (Abstract Recursive Functions or ARF). Henceforward we present definition of certain IR-functions using AR-functions.

AR-function arguments:
FB body function - represents body or "main" function of recusrsion
FA accessor function - returns current element
FR restrictor function - returns successive elements
A, B input parameters

ARF-2R, right folding recursion on two input arguments:

```
(LABEL ARF-2R
 (LAMBDA (FB FA FR A B)
   (IF A
       (FB (FA A B)
        (ARF-2R FB FA FR (FR A B) B))
       B)))
```

ARF-2L, left folding recursion on two input arguments:

```
(LABEL ARF-2L
 (LAMBDA (FB FA FR A B)
   (IF B
       (ARF-2L FB FA FR (FB A (FA A B))
        (FR A B))
       A)))
```

AR-functions may be also defined for more input arguments analogously. See [9] for details.

Definition of IR-functions using AR-functions: the definition of `FOLDR` and `FOLDR` is similar. The accessor and restriction function is almost identical, the only difference is that `ARF-FOLDR` is defined using `ARF-2R` and `ARF-FOLDL` using `ARF-2L` because of the folding direction of recursion. The definition of `ARF-MAP` is a bit more complicated. Body function is `(CONS A B)` because `MAP` conses results of application of accessor function to the arguments. The accessor function is `(FB (CAR A) B)` where `FB` represents a body function. This is because the body function is applied on every element of `A` and result is accumulated in `B`.

```
(LABEL ARF-FOLDR
 (LAMBDA (FB A B)
   (ARF-2R FB
           (LAMBDA (A B) (CAR A))
           (LAMBDA (A B) (CDR A))
           A B)))

(LABEL ARF-FOLDL
 (LAMBDA (FB A B)
   (ARF-2L FB
```

```
          (LAMBDA (A B) (CAR B))
          (LAMBDA (A B) (CDR B))
          A B)))

(LABEL ARF-MAP
 (LAMBDA (FB A B)
   (ARF-2R
    (LAMBDA (A B) (CONS A B))
    (LAMBDA (A B) (FB (CAR A) B))
    (LAMBDA (A B) (CDR A))
           A B)))
```

4.3 Evolving of Recursive Functions Using AR-functions

This subsection introduces several examples of the utilisation of AR-functions to recursive functions. Table 2 summarises search effort required to evolve `AR-FOLDR` and `AR-FOLDL` structure abstraction function. `ARF-2R` and `ARF-2R` was predefined in language function set.

objective	$I(1, i, 0.99)$
`AR-FOLDR`	28.6
`AR-FOLDL`	26.1

Table 2. Search effort required to evolve `AR-FOLDR` and `AR-FOLDL`

Example 1, `ADD`**:** let us imagine natural numbers as lists, where length of list denotes the value of number. For instance: `NIL` as 0, `(NIL)` as 1, `(NIL NIL)` as 2 and so forth. The `ADD` function appends input lists. It can be viewed as addition operation. For instance, `(ADD '(NIL NIL) '(NIL))` evaluates to `(NIL NIL NIL)`.

Recursive version: multiple objective function definition: main objective function: `ADD`, none sub-objective function.

```
(LABEL ADD
 (LAMBDA (A B)
   (IF A (CONS (CAR A) (ADD (CDR A) B)) B)))
```

ARF version: multiple objective function definition: main objective function: `ADD`, sub-objective function: `ARF-FOLDR`.

```
(LABEL ADD
 (LAMBDA (A B)
   (ARF-FOLDR (LAMBDA (A B) (CONS A B)) A B)))
```

Example 2, `SUB`**:** this function subtracts natural numbers represented as lists. For instance, `(SUB '(NIL NIL NIL) '(NIL))` evaluates to `(NIL NIL)`.

$$sub(a,b) = \begin{cases} a-b \text{ if } a > b \\ nil \quad \text{else} \end{cases} \tag{1}$$

Recursive version: multiple objective function definition: main objective function: `SUB`, none sub-objective function.

```
(LABEL SUB
 (LAMBDA (A B)
   (IF B (SUB (CDR A) (CDR B))) A))
```

ARF version: multiple objective function definition: main objective function: `SUB`, sub-objective function: `ARF-FOLDL`.

```
(LABEL SUB
 (LAMBDA (A B)
   (ARF-FOLDL (LAMBDA (A B) (CDR A)) A B)))
```

Example 3, `EP-N`**:** this function computes the even-n-parity problem. For instance, `(EP-N '(NIL NIL NIL T T NIL))` evaluates to `T`.

Recursive version: multiple objective function definition: main objective function: `EP-N`, sub-objective function: `XOR`.

```
(LABEL EP-N
 (LAMBDA (A B)
   (IF A
       (XOR (CAR A) (EP-N (CDR A) B))
       B)))
```

ARF version: multiple objective function definition: main objective function: `EP-N`, sub-objective function: `ARF-FOLDR`.

```
(LABEL EP-N
 (LAMBDA (A B)
  (ARF-FOLDR XOR A B))))
```

Let's see example 1 and example 3 which well demonstrate the advantages of structure abstraction. Both of the `ADD` and `EP-N` functions compute different problems, nevertheless the definitions using ARF are nearly identical (except `XOR`, `CONS`) due to abstracting their code structure into function `ARF-FOLDR`.

Example 4, `FACTORIAL`**:** multiple objective function definition: main objective function: `FACTORIAL`, sub-objective function: `ADD`, `MUL` (multiplication, see [9] for details). We present this function as an example where AR-functions are appropriate to define `FACOTRIAL` whereas IR-functions do not. However, even with AR-functions comes a complication. The `FACTORIAL` uses quite unusually two input arguments. The first argument represents input value and second argument, represent constant `'(NIL)`. This is due to `ARF-2R` returns `B` when `A` is empty list so that this captures a situation when factorial of zero is computed. Therefore `(FACTORIAL NIL '(NIL)` will evaluate to `'(NIL)`.

```
(FACTORIAL '(NIL NIL NIL) '(NIL)) => (NIL NIL NIL NIL NIL NIL)
```

```
(LABEL FACTORIAL
 (LAMBDA (A B)
   (ARF-2R
    (LAMBDA (A B) (MUL A B))
    (LAMBDA (A B) A)
    (LAMBDA (A B) (CDR A))
    A B)))
```

Table 3 presents results of reduction of search effort using AR-functions in comparison to recursive version. The ARF-powered experiment used apFSM language extended with `ARF-2R`, `ARF-2L`, `ARF-FOLDR` and `ARF-FOLDL` acquired as sub-objective functions. The table values are stated in thousands and does not include search effort required to evolve solution of `AR-FOLDR` and `AR-FOLDL`. The corresponding values are depicted in Table 2. For more detail about code structure abstraction and obtained results see [11].

objective	$I(1, i, 0.99)$, recursive	$I(1, i, 0.99)$, ARF
`EP-N`	2483.2	3.6
`ADD`	5948.8	3.4
`SUB`	13480.5	7.8
`FACTORIAL`	not found	32.6

Table 3. Comparison of search effort required to evolve recursive functions

5 Conclusion

This paper introduced functional paradigm based language apFSM for automatic programming of modular and recursive programs. We demonstrated our approach on the code structure abstraction of certain types of recursion. The results of experiments proved the advantages of structure abstraction on recursive functions in comparison to classical recursion. The results obtained on the

even parity problem outperform Koza's results with traditional GP [2], GP with ADF, GP with ARL [12], GP with MA [13] and PushGP [14].

References

1. Koza, J.R., Andre, D.: Evolution of iteration in genetic programming. In Fogel, L.J., Angeline, P.J., Baeck, T., eds.: Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming, MIT Press (1996)
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). The MIT Press (December 1992)
3. Kahrs, S.: Genetic programming with primitive recursion. In et. al., M.K., ed.: 8th annual conference on Genetic and evolutionary computation. Volume 1., ACM SIGEVO, ACM Press (July 2006) 941–942
4. Yu, T.: Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. Genetic Programming and Evolvable Machines **2**(4) (2001) 345–380
5. Salustowicz, R.P., Schmidhuber, J.: Probabilistic incremental program evolution: Stochastic search through program space. In van Someren, M., Widmer, G., eds.: Machine Learning: ECML-97. Volume 1224., Springer-Verlag (1997) 213–220
6. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–8
7. Steele, G.L.: COMMON LISP: the language. Digital Press, pub-DP:adr (1984) With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.
8. Dybvig, K.K.: The SCHEME programming language. Prentice-Hall, Inc. (1987)
9. Dostál, M.: A Functional Approach to Automatic Programming. PhD thesis, University of Hradec Králové, Hradec Králové, Czech Republic (2005) (written in Czech).
10. Hudak, P., Peterson, J., Fasel, J.: A gentle introduction to haskell 98 (1999)
11. Dostál, M.: On evolving of recursive functions using lambda-abstraction and higher-order functions. Logic Journal of the IGPL **13**(5) (2005) 515–524
12. Rosca, J.P., Ballard, D.H.: Genetic programming with adaptive representations. Technical Report TR 489, Rochester, NY, USA (1994)
13. Angeline, P.J., Pollack, J.B.: Evolutionary module acquisition. In Fogel, D., Atmar, W., eds.: Proceedings of the Second Annual Conference on Evolutionary Programming, La Jolla, CA, USA (25-26 1993) 154–163
14. Spector, L., Robinson, A.J.: Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines **3**(1) (2002) 7–40

Detecting Data Structures from Traces

Alon Itai and Michael Slavkin

Department of Computer Science
Technion Israel Institute of Technology
Haifa, Israel
itai@cs.technion.ac.il, mishaslavkin@yahoo.com

Abstract. This paper shows how to deduce the data structure that gave rise to a trace of memory accesses. The proposed methodology is general enough to classify different sets of data structures and is based on machine learning techniques.

1 Introduction

1.1 The Problem

The paper discusses the following problem:

> Given a *trace*, i.e., a sequence of address accesses to the data area of a program, detect what is the data structure that gave rise to the trace.

We assume that the set of possible data structures is given in advance. Also, since many data structures have several slightly different implementations (a linked list can be implemented with either a dummy header node or without it; it can also have a dummy end node) each distinct implementation is considered a distinct data structure. In this research we have limited ourselves to the case where the program uses a single static data structure, i.e. it only supports searches (no insertions/deletions) and all searches are to a single data structure.

Our thesis is that each data structure gives rise to a distinct pattern of traces, thus by classifying the traces one can recover the data structure that produced them. Machine learning algorithms were applied to perform the classification: In the *training phase* we extracted features from the traces and then passed them to the Machine Learning algorithm for classification. In the validation phase, we employed the resultant classification algorithm to classify new traces.

We have experimented with a number of popular classification algorithms (C4.5 [1], SVM [2], Naïve Bayes [3]) and compared their accuracy. The decision tree that results from the C4.5 classifier provides us also with an easy to understand algorithm for classifying data structures.

1.2 Motivation

Predicting the behavior of a program's use of memory is of high interest for different level tools, from operating systems and compiler optimization engines

to data structure implementations. They all can improve their performance given an effective algorithm for making such predictions.

(a) Current memory hierarchy solutions are based mostly on time and space locality of accesses and the LRU approach for ageing of cache lines (for associative caches) and memory pages. An algorithm for predicting the behavior of memory access at a high level of abstraction might provide much better solutions. For example, the Symmetrix Disk Array (of EMC) [4] used the patterns of accesses to prefetch elements of arrays and matrices.
(b) An operating system or a compiler can optimize memory allocation layout, given the feedback of an algorithm for detection of data structures from a previous run. For example, in case of hashing with chaining we would like to allocate nodes from the same chain at the same physical page. We may also try to activate the data structure detection algorithm on a prefix of the current run to achieve a performance improvement for the rest of the run. The JVM uses Just-in-Time compilation, i.e., the frequently executed parts of the JAVA programs are compiled dynamically, while the less frequently executed JAVA code is interpreted [5]. Thus, by the time of the dynamic compilation the compiler may utilize the feedback of the data structure detection algorithm on a prefix of the current run.
(c) The actual usage of generic data structures can be far from optimal for a specific insertion/deletion/search sequence. For example, choosing the STL class 'list' to implement a buffer can be a bad choice if the stored objects are not processed in LIFO or FIFO order. It is surely poor design to have chosen a list. However, if chosen we would like to have a methodology to detect such cases so that an inappropriate data structure could possibly be replaced online.

2 Problem Definition

"A data structure is a way to store and organize data in order to facilitate access and modifications" [6]. Data is organized as records each consisting of a sequence of fields. A field is either:

(a) a primitive type, or
(b) a pointer to a record, or
(c) one or more sub-records (allowing for recursion).

Following software engineering design principles, we assume that all data structures are encapsulated, i.e., all access to the data structures is via access functions. In particular every search initiates at the root of the tree, head of a list or at an arbitrary place in an array. Consequently, pointers that are not related to the implementation of the data structure are not allowed in this study.

Note that a data structure is characterized not only by its organization but also by its access pattern and the modifications it supports. For example, a heap can be implemented by an array, i.e., its organization is identical to the

organization of an array while its operations and hence its access pattern are different.

Since in this paper we restrict our study to searches only (no insertions or deletions) the data structure is static – its structure (topology of records and relations between them) doesn't change throughout the program.

We look at traces of memory accesses (address sequences) resulting from a series of searches to detect the underlying data structure organization and the traversal methods.

If we know the correspondence between an address and the record that resides at that address we can replace each address in the trace by some symbolic representation of the set of corresponding records. The resultant trace is called the *symbolic trace*.

We have thus defined two subproblems: learning the symbolic trace from the address trace, and learning the data structure associated with the symbolic trace. Decoupling the two problems sets a yardstick by which we can measure the effectiveness of the learning method, and the success in deducing the symbolic trace, as well as how are our results affected by the lack of knowledge about the layout structure of the records.

3 The Symbolic Trace

We first examine the symbolic trace and describe an approach to recover the topology of records and the relations between them, as well as the traversal methods.

The association between addresses and records is not one-to-one: an address can correspond to an entire record or its first subrecord. To distinguish between such cases, the symbolic trace is partitioned into levels: the top level being the records themselves, the next level is that of the subrecords etc.

For example, an address may be associated with a matrix entry, a row of the matrix and the matrix as a whole (if it is implemented as one long array). Thus we consider three traces: first level trace for matrices, second level trace rows and third level trace for the matrix elements.

We consider each level of record traces in isolation. Each level corresponds to a specific topology of the subset of records of this level and the subset of relationships between them, as well as the traversal methods. To detect the data structure we combine data from all levels.

Since the data structures are encapsulated, within a level the program can only move from one record to another via a pointer or from a subrecord to another subrecord (an important special case is moving from an array element to the next array element). Since a record and its subrecord belong to different levels, a symbolic trace of a single level does not include such moves.

For any two records A and B we would like to capture the relation "there is a pointer from A to B" from the symbolic trace. However, the trace manifests only the proximity of accesses to two records, which could arise from a sequential access to an array where adjacent elements will be accessed one after the other.

Since this access pattern is typical of arrays it may be used to characterize data structures.

To count how many times B appears immediately after A in the trace, we constructed the following binary matrix M:

$$M[A,B] = \begin{cases} 1 & \text{if } A \text{ immediately precedes } B \text{ at least } \theta \text{ times;} \\ 0 & \text{otherwise} \end{cases}$$

θ is a predetermined constant, which in our implementation was equal to 1.

For a record B let its *in-degree* be the number of records A for which $M[A,B] = 1$, and its *out-degree* the number of records C for which $M[B,C] = 1$. Let *in/out-degree* stand for edges in both directions, i.e., for the case where both (A,B) and (B,A) exist. Let us look at the distribution of tiplets of (in-degree, out-degree, in/out-degree) for records of several common data structures:

1. A binary search tree: the distribution consists of the tuples (1,1,0), (1,2,0) for nodes, (1,0,0) for leaves and (0, 2,0) or (0,1,0) for the root.
2. A deterministic skip list: we should have (0, many,0) for the $-\infty$ node, (many,0,0) for the $+\infty$ node and (1,2,0) for most of other nodes.
3. A singly linked list: we have (1,1,0), (0,1,0), (1,0,0).
4. A doubly linked list: the most common tuple is (2,2,2).

Thus the distributions of tuples differ significantly among data structure. Note that even for the same data structure the distribution of the tuples of a single search and multiple searches may differ. For example, for a binary search tree we expect the root to incur (0, 2, 0) or (0, 1, 0). In a trace of a sequence of several searches the in-degree of the root will be much greater than 0, because we return to the root to start the next search.

These degrees have a range from 0 to ∞. Since the exact value of any such degree is significant only when it is small but for large values we need only a qualitative measure, we partitioned the entire range to a finite number of subranges. The records are thus partitioned into equivalence classes depending on the values of the subranges of their (in-degree, out-degree, in/out-degree). We then replaced the occurrence of each record in the trace by its equivalence class.

To capture the time dependency, we looked at short subsequences in the trace. We limited their length to 3, since longer subsequences produced more features than the learning programs could handle, while limiting the length to 1 or 2 produced poor results. The learning of the data structure from the symbolic trace will be discussed in Section 5.

4 Address Trace

To deduce the symbolic trace from the address trace we use clustering algorithms. I.e., we first partitioned the addresses into clusters and then we examined cluster trace – the occurrence of the clusters in the trace.

As in the symbolic trace of Section 3 we would like to partition the cluster trace into *levels*. To achieve this we ran the clustering algorithm with different resolutions (the exact set of resolutions depends on the clustering algorithm and it will be discussed in the next paragraph). The problem of detecting a data structure has thus become a problem of detecting the topology of clusters and relations between clusters, and the traversal methods. As in Section 3 we converted this problem into a set of sub-problems, one for each clustering level. Then we combined data from several sources, one for each trace level, to detect the data structure.

We examined three clustering algorithms:

(a) Single-pass clustering [7] – a clustering technique that joins addresses A and B into the same cluster if the distance between these addresses is below some threshold. We may calculate such clusters by a single pass through a sorted list of addresses. We partitioned the distances into subranges whose sizes grow as 2^{2^n}.
(b) Agglomerative hierarchical clustering [8] — a clustering technique that starts with singleton clusters (with a single address per cluster) and then merges a number of pairs of clusters one by one. The two clusters with the best similarity measure are selected for merging. The similarity measure that we used is the distance between centroids (mean elements) of the clusters [9]. The number of merges is a function of the clustering resolution.
(c) Paging clustering – an approach that zeroes the least significant bits of an address to calculate the corresponding cluster (page). The number of bits zeroed depends on the resolution, i.e., the cluster (page) size and alignment vary for different clustering resolutions. Note, that the calculation of the cluster of an address is independent of other addresses. This clustering technique is very sensitive to the alignment of records in memory, but it is not aware of the actual distribution of memory accesses. Even though this clustering technique can be very inexact in some cases, it is still interesting to examine its performance because it takes only O(1) time to calculate to which cluster an address belongs.

We proceed with the cluster trace as we did with the symbolic trace.

5 Applying Machine Learning

We examined three popular classification approaches (C4.5, SVM and Naïve Bayes) to classify data structures. SVM and Naïve Bayes provide us only with a classification, while C4.5 results also in a decision tree, which helps us visualize the classification decisions. The common paradigm for the training phase is as follows:

(a) Prepare a set of training examples that contains a number of samples for each data structure.
(b) Convert each example into a set of features.

(c) Train the classifier on a set of pairs of the form (data_structure, set_of_feature_values) and prepare all the files that are necessary for the testing phase (these files constitute the learning model and they are specific to each classifier).

The common paradigm for the testing phase is as follows:

(a) Prepare a set of testing examples that contains a number of samples for each data structure.
(b) Convert each example into a set of features.
(c) Run the classifier on the set of features of each example and output the predicted data structure.

Both the training and the testing examples were generated artificially. To create examples we first generated a sample of the data structure with random parameters and random entry values. Then we generated a series of searches, which are typical for this type of data structure (e.g., searches to random keys that start at the root of the tree, at the head of a singly linked list, at an arbitrary place in an array and so on), applied to the current sample. We recorded the trace of the memory accesses (the address sequence) resulting from this series of searches. In the symbolic trace mode we recorded symbols (which encode information about all the records that correspond to the appropriate address) instead of addresses.

We collected tuples and treated them as words in the Bag-of-Words approach for text categorization [10]. In the Bag-of-Words approach the question of whether taking the existence of a feature or the number of occurrences is called *feature evaluation*. Taking the existence it is called *binary* and the number is called *natural*. Finally, as is customary, we take the logarithm of the number of occurrences instead of the occurrences .

6 The Experiment

6.1 The Setup

We examined classification approaches from the previous chapters for eight data structures (for each data structure we examined searches to several random keys that either belong or do not belong to the data structure):

(a) An AVL tree.
(b) A deterministic skip list.
(c) A doubly linked list with searches, which start from the first or from the last node at random.
(d) A singly linked list.
(e) A hash table with conflict resolution solved by chaining.
(f) A matrix where each search starts at some random displacement from the row (column) start and continues for a random number of steps with a random step length. The direction of the search is also random.

(g) A vector where each search starts at some random displacement from the row (column) start and continues for a random number of steps with a random step length. The direction of the search is also random. We examine two different cases of this data structure:
   i) A vector with a mainly random access pattern, which is produced by a high percentage of long steps (we will quickly go out of the vector boundary with such steps).
   ii) A vector with a mainly sequential access pattern, which is produced by a high percentage of short steps.

We covered all combinations of the next input parameters:

- The type of a classifier.
- The symbolic trace or the address trace with all the clustering algorithms.
- The binary vs. natural feature evaluation.
- The noise threshold.
- Clustering levels (resolutions) supported.

We generated 300 training examples and 300 testing examples per data structure and per each combination of input parameters. The generation of the data structure sample incurs the execution of several memory allocations that are produced by the compiler. It follows an allocation algorithm that might be biased. For example, the Microsoft Visual C++ compiler attempts to allocate all records sequentially in memory. This might lead to a bias: the addresses of consecutive elements in a list might form an arithmetic sequence, and thus will be indistinguishable from an array. For our experiment we tried to avoid such a bias by using of our own memory allocator, which chooses the next allocated address at random.

6.2 An Example

Figure 1 depicts two decision trees which were created by the C4.5 classifier with the binary feature evaluation and the single-pass clustering algorithm.

Fig. 1(a) depicts the decision tree that distinguishes between deterministic skip lists and AVL trees. Multiple searches to an AVL tree cause the root to have high in-degree. In deterministic skip lists both the root and the $+\infty$ node have high in-degree. Word a.2 describes accesses to two distinct nodes with large in-degrees. This can occur only for deterministic skip lists.

Fig. 1(b) depicts the decision tree that distinguishes between doubly linked lists and AVL trees. Nodes (or clusters of resolution 1) with (in-degree, out-degree) = (1, 2) are typical of AVL trees but not of doubly linked lists. If no such a sequence exists (word b.1) we obviously have a doubly linked list.

6.3 Performance

In Figure 2 we compared the accuracy of classifiers in conjunction with different clustering approaches. These clustering approaches included the three clustering

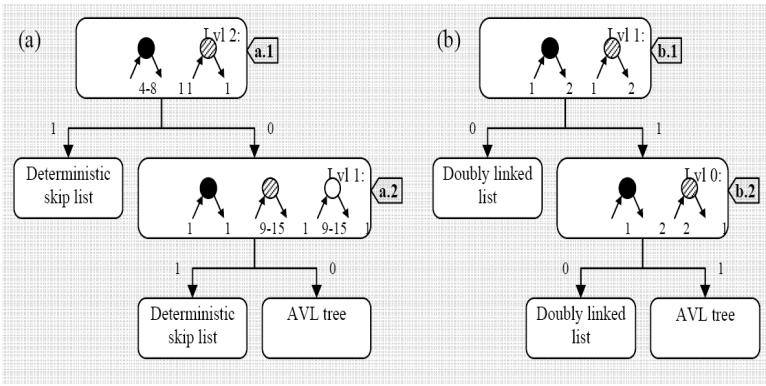


Fig. 1: Decision trees for distinguishing between (a) deterministic skip lists and AVL trees (b) Doubly linked list and AVL tree

algorithms of Section 4 for the address trace and the symbolic trace approaches. Moreover, for the same clustering algorithm we looked at two different subsets of clustering levels. One subset (clusters 0 to 3) contained the highest resolution (zero resolution) which spreads each address per cluster while the second (cluster 1to 3) did not contain it. The remaining resolutions appeared in both subsets.

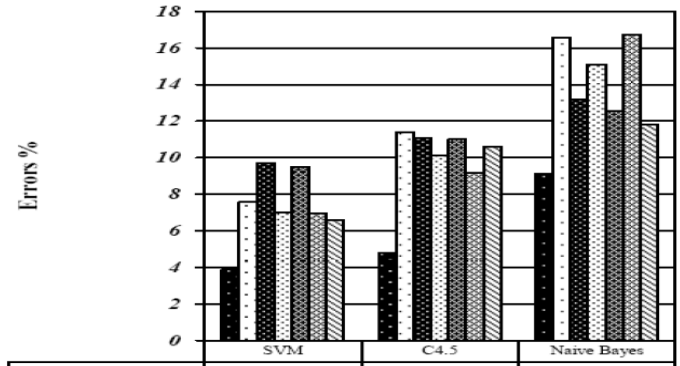
Note the relatively poor performance of Naïve Bayes approach. The Naïve Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent. In our case, there are short words that appear as parts of longer words. Hence their attribute values are dependent.

As can be seen, the symbolic approach provides us with the best classification accuracy for SVM and C4.5 classifiers. There is not much difference in the performance of the remaining clustering algorithms. It is especially interesting that the accuracy of the paging approach (whose cluster calculation is easy) is in line with other clustering techniques, which take into account the actual distribution of memory accesses.

Note that in most of the cases the clusters 0–3 subset of clustering levels results in a better accuracy than the clusters 1–3 subset. In most of the data structures that we examined the record size is greater than one and there is a specific access pattern for primitive fields of the same record. We miss this access pattern if we do not look at words from the highest resolution level (with a single address per cluster).

The performance of the binary feature evaluation case is very similar to the natural one. We have not included the graphs due to lack of space.

In Table 1 we present the accuracy of the SVM classifier tested on a symbolic trace with the natural feature evaluation. We plot the distribution of actual data structures at a testing phase vs. predicted ones. It should be noted that most of



	SVM	C4.5	Naive Bayes
symbolic	3.875	4.8	9.125
address, clusters0to3, hierarchical	7.58333	11.4	16.5833
address, clusters1to3, hierarchical	9.708333	11.1	13.1667
address, clusters0to3, onepass	7	10.1	15.0833
address, clusters1to3, onepass	9.45833	11	12.5417
address, clusters0to3, paging	6.95833	9.2	16.75
address, clusters1to3, paging	6.58333	10.6	11.7917

Fig. 2: Natural feature evaluation with different traces (symbolic or address in conjunction with the different clustering algorithms)

the errors occur between vectors with a random access pattern and vectors with a sequential access pattern. The difference between definitions of these two data structures is indeed fuzzy.

6.4 Robustness

In the current research we addressed the case where the program uses a single data structure. In the real world accesses to a number of different data structures are interleaved within a single program. It is important to examine the robustness of our technique to the addition of noise. In Figure 3 we compare the robustness per classifier and per clustering algorithm. As it can be seen, all (classifier, clustering algorithm) pairs are very sensitive to noise.

6.5 Statistical Significance of the Results

A statistical analysis of the results shows that with over 95% confidence the error is under 1%.

Actual Predicted	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
(a) AVL tree	297	1	0	0	1	1	0	0
(b) Determinsitc skip list	0	297	0	3	0	0	0	0
(c) Doubly linked list	0	0	291	9	0	0	0	0
(d) Singly linked list	0	0	0	300	0	0	0	0
(e) Chained hashing	0	0	0	7	293	0	0	0
(f) Matrix	0	1	0	0	5	294	0	0
(g) Randomly Accessed Vector	0	0	0	0	3	0	279	18
(h) Sequentially Accessed Vector	0	0	0	0	0	0	44	256

Table 1. Accuracy of SVM on a symbolic trace and natural feature evaluation

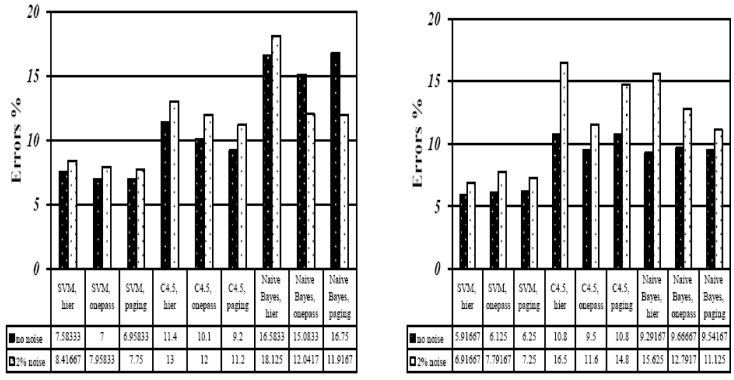


Fig. 3: The robustness per classifier and per clustering algorithm
using the clusters0to3 subset of clustering levels.
On the left the natural feature evaluation; on the right the binary one.

7 Concluding Remarks

In this study we have described the machine learning approach to classifying any subset of data structures. We have considered the case where the program uses a single data structure, which only supports searches. Scaling this technique to data structures, which support insertions/deletions, and to the case of multiple data structures is left for further studies. Our technique is easily extendable to additional data structures and implementations.

Our statistical measurements show the high quality of classification in the absence of noise, especially when using the SVM and C4.5 classifiers. On the other hand, the approach to feature selection described in our solution is very sensitive to noise. We will have to overcome this lack of robustness in order to proceed to real world applications with accesses to a number of different data structures interleaved within a single program.

When we used the C4.5 classifier we got a classification algorithm (decision tree) for every subset of data structures. The time required to classify the trace of memory accesses into a data structure is $O(t)$, where t is the total time to calculate all the features appearing on the appropriate path in the tree. Thus we considered techniques to reduce the time spent on the calculation of a single feature without the accuracy degradation, such as:

- The usage of the quick paging clustering model.
- The usage of the binary feature evaluation, instead of the natural one (in this case we need not to search the entire trace).

Let us conclude with directions for the future research:

(a) At the current stage we examined the case of static data structures, i.e., only searches are supported (no insertions/deletions). To scale up this technique to dynamic data structures we need also to consider the type of the memory access (read or write). We can add this information as an additional entry to our tuple.

(b) In general, memory accesses may be generated by a number of sources. Moreover, sometimes several data structures are processed simultaneously (e.g., matrix multiplication).
   i) Sometimes our techniques can still be applied if we change the definition of the data structure. For example, we may consider a pair of multiplied matrices as a single data structure.
   ii) If several data structures are processed sequentially we may also consider contributions of both time and code locality.

(c) The robustness of our approach has to be improved in order to proceed to real world applications that access a number of different data structures interleaved within a single program.

(d) In addition we should take into account the time required to calculate the features. Thus we could gain both accuracy and classification time by better feature reduction.

References

1. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
2. Vapnik, V.: The Nature of Statistical Learning Theory. Springer-Verlag, New York (1995)
3. Mitchell, T.: Machine Learning. McGraw Hill (1997)
4. EMC Corporation: Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide. (1999)
5. Ebcioglu, K., Altman, E., Hokenek, E.: A Java ILP machine based on fast dynamic compilation. In: IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java. (1997)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company (1989)
7. Rieber, S., Marathe, U.P.: The single pass clustering method. Technical Report ISR-16, Cornell University, the Department of Computer Science (1969) to the NDF.
8. Johnson, S.: Hierarchical clustering schemes. In: Psychometrika. Volume 32. (1967) 241–254
9. Sokal, R., Michener, C.: A statistical method for evaluating systematic relationships. The University of Kansas Scientific Bulletin **38** (1958) 1409–1438
10. Salton, G., McGill, M.: Introduction to Modern Information Retrieval. McGraw-Hill Book Company (1984)

Author Index