Inductive Synthesis of Logic Programs by Composition of Combinatory Program Schemes*

Andreas Hamfelt†
Computing Science Department
Uppsala University

Jørgen Fischer Nilsson
Department of Computer Science
Technical University of Denmark

May 25, 1998

Abstract

Based on a variable-free combinatory form of definite clause logic programs we outline a methodology and supporting program environment for inducing logic programs from examples. The combinators comprise *fold* combinators for recursion on lists. The combinator form avoids confusing object and metavariables in the applied metalogic program environment. Moreover, the combinators enable piecewise composition of semantically meaningful program elements according to the compositional semantics principle. In addition useful algebraic rewriting rules can be formulated conveniently with the combinators.

Keywords: logic program schemata, logical combinators, synthesis by composition and specialization of schemas, inductive synthesis, metalogic program environment.

1 Introduction

Induction of logic programs (ILP) is traditionally conducted as a generalization process in which program clauses are synthesized by generalization involving anti-unification from input/output examples [10]. This approach conforms with induction of logical theories as carried out in the machine learning tradition. It has turned out to be difficult to synthesize recursive program structures (i.e., programs applying recursively defined predicates) from sample facts with no *a priori* indication of recursive structure. In [5] we tried to tackle this problem by suggesting that the program structure to be induced has to comply with given recursion schemes of universal nature. The induction process then essentially reduces to synthesizing of auxiliary argument predicates. The auxiliary predicates in simpler cases would be defined by non-recursive clauses formed by a least general generalization procedure from the examples. In more complex cases the whole induction procedure may be invoked recursively to invent a new recursively defined auxiliary predicate. The proposed framework METAINDUCE is established using a metalogic program setting.

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

†Supported by a grant from Volvo Research Foundation, Volvo Educational Foundation and Dr Pehr G. Gyllenhammar's Research Foundation.

In a number of more recent papers [6, 7, 8] we have introduced a novel combinatory form of logic programs enabling variable-free formulations of definite clauses. This paper addresses the usefulness of the combinator form of programs for ILP.

2 Compositional Logic Programming

In the structured programming tradition programs are composed from specifications in a top-down problem decomposition manner according to which a larger program is obtained by putting together smaller programs by means of combining forms. In functional programming functional composition offers itself as one of the forms together with conditionals and certain recursion forms [1]. In logic programming it is more difficult to identify combinators since there are no standard combining forms for n-ary predicates.

In a recent paper [8] we pursue the compositional principle in the context of logic programming by proposing a variable-free combinator form of definite clauses termed COMBILOG. Consider a collection of defining definite clauses for a predicate p

$$p(\vec{X}) \leftarrow \bigvee_{i}^{\gamma} \bigwedge_{j}^{\beta_i} q_{ij}(\vec{t_{ij}})$$

where $\vec{X}$ are tuples of variables and $\vec{t}$ are tuple terms conforming with the arity of the predicates. This is replaced by

$$p \leftarrow \phi$$

where ϕ is a ground combinatory predicate term of the form

$$\langle cterm \rangle ::= \langle pid \rangle \mid \langle comb \rangle (\langle cterm \rangle \{, \langle cterm \rangle\}^*)$$

where $\langle pid \rangle$ is a predicate identifier and $\langle comb \rangle$ is one of the three combinators *and*, *or*, and $make[\mu_1, \ldots, \mu_m]$, where μ_i are index numbers. These three combinators are identified as the least semantically meaningful building blocks being sufficient and necessary in our analysis. The combinators can be defined as follows in the informal use of higher order predicates.

$$
\begin{aligned}
&or(P,Q)(X_1,\ldots,X_n) \leftarrow P(X_1,\ldots,X_n).\\
&or(P,Q)(X_1,\ldots,X_n) \leftarrow Q(X_1,\ldots,X_n).\\
&and(P,Q)(X_1,\ldots,X_n) \leftarrow P(X_1,\ldots,X_n) \wedge Q(X_1,\ldots,X_n).\\
&make[\mu_1,\ldots,\mu_m](Q)(X_{\mu_1},\ldots,X_{\mu_m}) \leftarrow Q(X_1,\ldots,X_n)
\end{aligned}
$$

As it appears *make* is a generalized projection and permutation combinator.

Here P and Q are predicate arguments and the combinators may be conceived as higher order predicates used similarly to higher order functions in functional programming. These combinators are of generic arity and in addition *make* is indexed with m distinct positive index numbers. These combinators are inspired by a proposal by Quine [12] for a variable-free form of full first order predicate logic, see [8] for a discussion.

The predicate identifiers include, besides user introduced predicate names, the four predefined predicates *cons*, *true*, *id*, $const_c$ defined by the following unit clauses

$cons(U, V, U.V)$
$id(X, X)$
$true$
$const_c(c)$ for all relevant constants c.

In [8] is proven combinatory completeness relative to definite clauses of these combinators in a canonical form with the defining term

$$or^*(make[\vec{\mu_i}](and^*(make[\vec{\mu_{ij}}]q_{ij})))$$

where or^* is zero, one or more applications of *or* and analogously for *and*.

The combinatory form of clauses is reflected in the compositional semantics so that the relational denotation of the term $comb(\phi_1, \ldots, \phi_n)$ is determined as a function of the relational denotations $[\![\phi_i]\!]$ of the subterms ϕ_i in accordance with the following semantical definitions of the combinators

$$\begin{aligned}
[\![and(\varphi, \psi)]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\
[\![or(\varphi, \psi)]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] \\
\llbracket make[\mu_1, \ldots, \mu_m](\phi)\rrbracket &= \{\langle t_{\mu_1}, \ldots, t_{\mu_m}\rangle \in H^m \mid \exists \langle t_1, \ldots, t_n\rangle \in [\![\phi]\!]\}
\end{aligned}$$

or more explicitly for *make*

$$\begin{aligned}
\llbracket make[\mu_1, \ldots, \mu_m](\phi)\rrbracket = \{&\langle t'_1, \ldots, t'_m\rangle \in H^m \mid \exists \langle t_1, \ldots, t_n\rangle \in [\![\phi]\!] \\
&\text{such that for } i = 1..m \text{ if } \mu_i \leq n \text{ then } t'_i = t_{\mu_i}\}
\end{aligned}$$

where H is the Herbrand universe of the program. Thus $make[1, 2, \ldots, n]$ is the identity combinator for n-ary relations, and the term $make[1, 2, \ldots, n](true)$ yields the n-ary universal relation from the 0-ary *true*.

These definitions are proved in [8] to make the semantics coincide with the usual least model and fix point semantics of definite clauses.

The denotations of the predefined predicates are as follows

$$\begin{aligned}
[\![id]\!] &= \{\langle t, t\rangle \in H^2 | t \in H\} \\
[\![cons]\!] &= \{\langle t', t'', t'.t''\rangle \in H^3 \mid t', t'' \in H\} \\
[\![true]\!] &= \{\langle\rangle\} \\
[\![const_c]\!] &= \{\langle c\rangle\} \text{ for all relevant constants } c
\end{aligned}$$

Thus, in the construction of the least fixed point for a combinatory form of program the iteration step reduces to set union, intersection and generalized projection.

3 Recursion Combinators

In the combinator language COMBILOG summarised above recursion is handled like in ordinary logic programs by introduction of predicates defined in terms of themselves. One way of eliminating these recursive definitions would be to introduce a general fix point combinator (cf. the Y combinator in combinatory logic). However, we prefer to introduce list-oriented recursion operators analogously to functional programming practice [2, 13]. In [6, 7] we propose two fold combinators for list recursion defined e.g., as follows.

$$\begin{aligned}&\mathit{foldr}(P,Q)(Y,\mathit{nil},Z) \leftarrow Q(Y,Z)\\&\mathit{foldr}(P,Q)(Y,X.T,W) \leftarrow \mathit{foldr}(P,Q)(Y,T,Z) \wedge P(X,Z,W).\end{aligned}$$

$$\begin{aligned}&\mathit{foldl}(P,Q)(Y,\mathit{nil},Z) \leftarrow Q(Y,Z)\\&\mathit{foldl}(P,Q)(Y,X.T,W) \leftarrow P(X,Y,Z) \wedge \mathit{foldl}(P,Q)(Z,T,W).\end{aligned}$$

which are <u>relational</u> counterparts of the well known functional operators [1, 2, 13]. Similar recursion operators are proposed in [4] applying λ-Prolog, whereas our combinators are made available in ordinary logic programming as explained in the next section.

For instance the relational *append* predicate definition in the extended COMBILOG becomes

$$\mathit{append} \leftarrow \mathit{make}[2,1,3](\mathit{foldr}(\mathit{cons},\mathit{id}))$$

where $\mathit{make}[2,1,3]$ swaps the two first arguments. The short and elegant combinatory formulations of *append* is seductive since combinatory formulations are admittedly sometimes awkward. For instance the unit clause

$$\mathit{make\_unit\_list}(X, X.\mathit{nil})$$

via the reformulation

$$\mathit{make\_unit\_list}(X,Y) \leftarrow \mathit{const}_{nil}(Z), \mathit{cons}(X,Z,Y)$$

which serves to remove non-variable terms, yields the combinatory definition

$$\mathit{make\_unit\_list} \leftarrow \mathit{make}[1,3](\mathit{and}(\mathit{make}[2,1,3](\mathit{const}_{nil}), \mathit{cons}))$$

being in the canonical form mentioned above.

In [7] we prove that logic program recursion with the fold schemes suffices for expressing all partial and general recursive functions. However, this is a purely theoretical result; in logic programming practice non-linear recursion forms such as *and-or* problem reduction schemes are convenient, cf. [3]. Still the empirical investigation in [7] tells that most common pure logic programs can be naturally formulated within the fold schemes; therefore we confine ourselves to the fold schemes in the present context.

Actually in [7] we state and in [6] we prove the following duality theorems

$$\mathit{foldr}(P,Q) \equiv \mathit{make}[3,2,1](\mathit{foldl}(\mathit{make}[1,3,2](P), \mathit{make}[2,1](Q)))$$

$$\mathit{foldl}(P,Q) \equiv \mathit{make}[3,2,1](\mathit{foldr}(\mathit{make}[1,3,2](P), \mathit{make}[2,1](Q)))$$

connecting the two fold operators in the declarative, that is model theoretic or fix point, understanding of logic programs.

For logic program expressible with the fold combinators, in the combinator formulation of such programs it is now unnecessary to introduce user specific predicate names so that such a program takes form of ground combinator term comprising only the predefined predicates.

4 Synthesizing Programs

In this section we sketch a methodology and programming environment for conducting inductive synthesis of logic programs in the pure combinator form

$$\langle cterm \rangle ::= \langle comb \rangle (\; [\; \langle cterm \rangle \; \{ \; , \; \langle cterm \rangle \; \}^* \;] \;)$$

where predicates (pre-defined) are conceived as 0-ary primitive combinators and $\langle comb \rangle$ includes, besides *and*, *or* and the indexed *make*, the operators *foldl* and *foldr*.

The program synthesis environment is established as a metalogic program where the combinator terms appear as ordinary terms as explained in [7, 8, 14], by way of a generic predicate *apply* as exemplified with *and*

$$\begin{array}{l}
apply(and(P,Q), X_1) \leftarrow apply(P, X_1) \wedge apply(Q, X_1) \\
apply(and(P,Q), X_1, X_2) \leftarrow apply(P, X_1, X_2) \wedge apply(Q, X_1, X_2) \\
\quad \vdots \\
apply(and(P,Q), X_1, \ldots, X_n) \leftarrow apply(P, X_1, \ldots, X_n) \wedge apply(Q, X_1, \ldots, X_n)
\end{array}$$

up to some fixed maximum arity n. The *apply* predicates enable execution of combinatory programs in the metalogic programming environment. The resulting semantics is proved equivalent in [8] to the semantics described above for the combinators.

During the course of the induction the combinator terms being synthesized may be non-ground, the variables representing unknown (sub)combinator terms. This means that our induction procedure proceeds by specializing (instantiating) partially combinatory program forms in contrast to the traditional induction by generalization from ground facts.

The advantage of the combinatory program form in the synthesis processes is that the absence of the object program variables eliminates confusion with the metavariables whose range are combinatory terms, i.e., subprograms. Moreover the strict compositional semantics ensures that subprograms represented by combinators can be synthesized independently of the embedding program context. This is in contrast to ordinary logic programs where a clause has to be completed before it represents a semantically meaningful entity.

The combinator term to be synthesized fulfils the following grammar

$$\begin{array}{lll}
\langle cterm \rangle & ::= & make[i_1, \ldots, i_n](\langle prepred \rangle) \mid \\
 & & make[i_1, \ldots, i_n](foldr(\langle cterm \rangle, \langle cterm \rangle)) \mid \\
 & & make[i_1, \ldots, i_n](and(\langle cterm \rangle, \langle cterm \rangle)) \\
\langle prepred \rangle & ::= & cons \mid id \mid true \mid const
\end{array}$$

corresponding to the above canonical form enriched with the *foldr* combinator. The identity combinator $make[1, 2, \ldots, n]$ is left implicit in terms.

The *or* operator is omitted since we have the following equivalences

$$foldr(or(P_1, P_2), Q) \equiv or(foldr(P_1, Q), foldr(P_2, Q))$$

$$foldr(P, or(Q_1, Q_2)) \equiv or(foldr(P, Q_1), foldr(P, Q_2))$$

which entails that a program requiring the *or* combinator can be *a priori* decomposed into *or*-free programs since *or* can distributed over *foldr* in both arguments and hence can be propagated to the top.

In principle the synthesis can now be carried out as a blindfold search for the shortest term of this grammar which complies with the suite of test examples. In order to boost the induction process we suggest that mode constraints be imposed so that search is pruned in the top-down construction of the program term as soon as the mode constraint turns out to be violated. The mode constraints are critical to our induction approach since absence of the constraints would admit the universal relation as simplest solution in absence of negative examples. Let us introduce the notion of *valence* for the combined constraints of modes, types and arity, assuming synthesis of well-moded programs only [6]. The intuition is that composition of combinators are subject to constraints like valence constraints on chemical atoms and radicals.

The heart of the synthesizer are defining clauses for a mutually recursive predicates *syn* and *syn_make* drawing on the above defining clauses for the combinators. This program structure is to ensure that combinators obey the above grammar. For instance for *and* and *foldr* there are the following clauses

$$\begin{array}{l} syn(and(P,Q),\ Valencel,\ Exlist) \leftarrow \\ \quad valence(and, Valencel, Valencel1, Valencel2), \\ \quad syn\_make(P,\ Valencel1,\ Exlist), \\ \quad syn\_make(Q,\ Valencel2,\ Exlist) \end{array}$$

$$\begin{array}{l} syn(foldr(P,Q),\ Valencel,\ Exl) \leftarrow \\ \quad valence(foldr,\ Valencel,\ Valencel1,\ Valencel2), \\ \quad split\_example\_foldr(Exl,\ Exl1,\ Exl2), \\ \quad syn\_make(P,\ Valencel1,\ Exl1), \\ \quad syn\_make(Q,\ Valencel2,\ Exl2) \end{array}$$

The predicate *valence* establishes relationship between the valencies of the combinator term and valencies of its predicate arguments. Similarly the predicate *split_example_foldr* calculates a list of examples for the predicate arguments on basis of the given list of examples. The tuples in the resulting examples are not necessarily ground.

The mode notation is rather self explanatory, e.g, the modes for *id* below require that either the second or first argument (or both) be ground upon invocation of *id*.

$$\begin{array}{l} syn(id, Valencel, Exlist) \leftarrow \\ \quad check\_valence(Valencel, [[*,+],[+,*]]), \\ \quad test\_examples([X,X], Exlist) \\ syn(cons, Valencel, Exlist) \leftarrow \\ \quad check\_valence(Valencel, [[+,+,*],[*,*,+]]), \\ \quad test\_examples([X,Y,[X|Y]], Exlist) \\ syn(const_c, Valencel, Exlist) \leftarrow \\ \quad check\_valence(Valencel, [*]), test\_examples([c], Exlist) \\ syn(true, [], \_). \end{array}$$

$$\begin{array}{l} syn\_make(make(Il,P),\ Valencel,\ Exl) \leftarrow \\ \quad make\_valence(Il,\ Valencel, Valencel1), \end{array}$$

$make\_example(Il, Exl, Exl1)$,
$syn(P, Valencel1, Exl1)$

$test\_examples(F, [])$.
$test\_examples(F, [F|L]) \leftarrow test\_examples(F, L)$

The top level is

$$synthesize(P, Valencel, Exl) \leftarrow syn\_make(P, Valencel, Exl).$$

It is a key feature of this approach that valence specifications *Vall* all have to be satisfied by the synthesized program. For instance when synthesizing the program $make\_unit\_list(X, [X])$ from examples, the two modes $[+, *], [*, +]$ would be specified.

5 Conclusions

We have outlined principles for inductive synthesis of logic programs based on composition of building blocks in the form of logical combinators. Our approach carries out the synthesis as a top down composition of the program constructions without generalization of program elements from examples. The combinatorial explosion inherent in the composition of combinators is tackled by imposing various constraints on the composition: (1) The resulting combinator program obeys a grammar for the combinators (2) Valence constraints comprising mode, type and arity constraints are to be fulfilled in each synthesis step. In addition to these general constraints various heuristics may be taken advantage of, for instance $[\![and(P,Q)]\!] \subseteq [\![P]\!]$. This combinatory approach is distinguished from MetaInduce [5] by avoiding the complications of object variables in synthesized clauses.

References

[1] J. Backus: Can Programming be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs. *Comm. of the ACM*, 21, 8, 1978. pp. 613–641.

[2] R. Bird & O. de Moor: *Algebra of Programming*, Prentice Hall, 1997.

[3] P. Flener: Inductive Logic Program Synthesis with DIALOGS. In [9] pp. 175–198.

[4] Gegg-Harrison, T. S.: Representing Logic Program Schemata in λProlog. In L. Sterling (ed.) *Procs. Twelfth International Conference on Logic Programming 1995*, MIT Press, London, 1995. pp. 467–481.

[5] A. Hamfelt & J. Fischer Nilsson: Inductive metalogic programming. In S. Wrobel (ed.) *Procs. Fourth International Workshop on Inductive Logic programming* (ILP-94), Bad Honnef/Bonn, GMD-Studien Nr. 237, 1994. pp. 85–96.

[6] A. Hamfelt & J. Fischer Nilsson: Declarative Logic Programming with Primitive Recursive Relations on Lists. In M. Maher (ed.) *Procs. Joint International Conference and Symposium on Logic Programming 1996*, MIT Press, London, 1996, pp. 230–242.

[7] A. Hamfelt & J. Fischer Nilsson: Towards a Logic Programming Methodology Based on Higher-order Predicates. *J. New Generation Computing*, vol 15, no 4, 1997. pp. 421–448.

[8] A. Hamfelt, J. Fischer Nilsson & A. Vitoria: A Combinatory Form of Pure Logic Programs and its Compositional Semantics. Submitted for publication 1998.

[9] S. Muggleton (ed.): Inductive Logic Programming, Proceedings of the 6th International Workshop on Inductive Logic Programming, 1996, *Lecture Notes in Artificial Intelligence* 1314, Springer, 1996.

[10] S.-H. Nienhuys-Cheng, R. de Wolf: Foundations of Inductive Logic Programming, *Lecture Notes in Artificial Intelligence* 1228, Springer, 1997.

[11] J. Fischer Nilsson & A. Hamfelt: Constructing Logic Programs with Higher Order Predicates. In M. Alpuente & M. I. Sessa (eds.) *Procs. Joint Conference on Declarative Programming 1995* (GULP-PRODE'95), Universita' Degli Studi di Salerno, Salerno, 1995. pp. 307–312.

[12] W. V. Quine: Predicate-functor Logic. In J. E. Fenstad (ed.) *Procs. Second Scandinavian Logic Symposium*, North-Holland, 1971. pp. 309–315.

[13] C. Reade: *Elements of Functional Programming*, Addison-Wesley, 1989.

[14] D. H. D. Warren: Higher-order extensions to PROLOG: are they needed ? In D. Michie (ed.) *Machine Intelligence 10*, Ellis Horwood and Edinburgh University Press, 1982. pp. 441–454.