Explanation-based generalisation
=
Partial evaluation

Frank van Harmelen
Alan Bundy

DAI Research Paper No. 347

Explanation-based generalisation
=
Partial evaluation[1]

Frank van Harmelen
Alan Bundy
Department of A.I.
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

Abstract

We argue that explanation-based generalisation as recently proposed in the machine learning literature is essentially equivalent to partial evaluation, a well known technique in the functional and logic programming literature. We show this equivalence by analysing the definitions and underlying algorithms of both techniques, and by giving a Prolog program which can be interpreted as doing either explanation-based generalisation or partial evaluation.

1 Introduction

An interesting development in the field of machine learning is the advent of a technique called *explanation-based generalisation* (EBG). This name was first coined in [Mitchell *et al.*, 1986], but the technique can be traced back to [Mitchell, 1983], and earlier to [DeJong, 1981] and [Mitchell, 1982]. This technique tackles the problem of formulating general concepts on the basis of specific training examples. For some considerable time, the functional programming community, and more recently the logic programming community, has been discussing a technique called *partial evaluation* (PE) as a program optimisation method (see [Futamura, 1971] for an early paper on PE, [Ershov, 1982] for PE in functional programming, and [Komorowski, 1982], [Venken, 1984], and [Takeuchi & Furukawa, 1986] for PE in logic programming). This paper shows that, in the context of logic programming, the two techniques, although developed for different purposes, do in fact consist of the same algorithm, and can both be implemented by (almost) the same piece of code. This close resemblance between EBG and PE has not been noted before, and indeed, a number of papers (such as [Prieditis & Mostow, 1987]), apparently treat EBG and PE as complementary techniques to achieve different goals, whereas in fact the two techniques are the same, as argued in this paper.

The rest of the paper is organised as follows: In section 2 and 3 we give brief descriptions of both EBG and PE. Section 4 paraphrases the definition of EBG to show that PE and EBG are essentially equivalent. Section 5 gives small implementations of both PE and EBG, and by comparing the two programs we see again that the two techniques are almost the same. Section 6 discusses the issue of guided versus unguided search that arises from the comparison of EBG and PE, and finally section 7 gives a worked example often used in the EBG literature to show how the computation corresponds to PE.

---

2   Explanation-based generalisation

EBG is a technique to formulate general concepts on the basis of a specific training example. The EBG algorithm consists of two stages. In the first stage EBG analyses a single training example in terms of knowledge about the domain and the goal concept under study, and produces an explanation of why the training example is an instance of the goal concept. The resulting explanation-structure is then used as the basis for formulating the general concept definition by generalising this explanation, i.e. abstracting it from the particular training example.

As input, the EBG algorithm expects the following items:

- Goal concept:
  A definition of the concept to be learned.

- Training example:
  A specific instance of the goal concept.

- Domain theory:
  A set of rules to be used in explaining why the training example is an instance of the goal concept

- Operationality criterion:
  A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed. This criterion defines a set of easily evaluated predicates from the domain theory.

Given these four inputs, the task is to determine a generalisation of the training example that is a sufficient definition for the goal concept and that satisfies the operationality criterion. Re-expressing the goal concept in these terms will make it operational with respect to the task of efficiently recognising examples of the concept. It is assumed that the input-definition of the goal concept does not satisfy the operationality criterion.

As mentioned above, the EBG algorithm is usually described as consisting of two stages:

1. Explain:
   Construct an explanation in terms of the domain theory that shows how the training example satisfies the goal concept definition. This explanation must be constructed so that each branch of the explanation structure terminates in an expression that satisfies the operationality criterion.

2. Generalise:
   Determine a set of sufficient conditions under which the explanation holds, stated in terms that satisfy the operationality criterion. This is accomplished by regressing (back propagating) the goal concept through the explanation structure. The conjunction of the resulting expressions constitutes the desired concept definition. This is typically done using a modified version of the goal-regression algorithm described in [Waldinger, 1977] and [Nilsson, 1980].

These two steps can be summarised as follows: the first step creates an explanation structure that separates the relevant feature values of the input example from the irrelevant ones. The second step analyses this explanation structure to determine the particular constraints on these feature values that are sufficient for the explanation to apply in general.

If the language used for expressing the input predicates and rules is logic, then the first step (explanation) amounts to proving that the input example is indeed an instance of the goal concept, in such a way that all the leaves of the proof tree

are operational predicates, while the second step consists of building a more general version of this proof that does not depend on any of the irrelevant feature values of the input example.

3   Partial evaluation

The main goal of PE is to perform as much of the computation in a program as possible without depending on any of the input values of the program. The theoretical foundation for PE is Kleene's S-M-N theorem from recursive function theory [Kleene, 1962, page 342]. This theorem says that given any computable function f of n variables, $f = f(x_1, \ldots, x_n)$, and k ($k \leq n$) values $a_1, \ldots, a_k$ for $x_1, \ldots, x_k$, we can effectively compute a new function f' such that

$$f'(x_{k+1}, \ldots, x_n) = f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n)$$

The new function f' is a specialisation of f, and is easier to compute than f for those specific input values. The PE algorithm can be regarded as the implementation of this theorem, and is in fact slightly more general in the context of logic programming: it allows not only that a number of input variables are instantiated, but also that these variables can be only partially instantiated to terms that contain nested variables. Furthermore, the PE algorithm allows k in the above theorem (the number of instantiated input variables), to be 0, that is, no input to f is specified at all. Even in this case the PE algorithm is often able to produce a definition of f' which is equivalent to f but more efficient, since all the computations performed by f that are independent of the values of the input variables can be precomputed in f'. Thus, the PE algorithm takes as its input a function (program) definition, together with a partial specification of the input of the program, and produces a new version of the program that is specialised for the particular input values. The new version of the program will then be less general but more efficient than the original version.

The PE algorithm works by symbolically evaluating the input program while in the mean time trying to 1) propagate constant values through the program code, 2) unfold procedure calls, and 3) branching out conditional parts of the code. If the language used to express the input program is logic, then the symbolic evaluation of the program becomes the construction of the proof tree corresponding to the execution of the program.

As mentioned, a special case of PE is when none of the values for the input variables $x_1, \ldots, x_k$ are given (in other words, $k = 0$). In this case, the PE algorithm cannot do as much optimisation of the input program, and as a result the new program will not be as efficient. However, the new program is no longer only a specialisation of the original program, but indeed equivalent to it. Thus, in this way PE can be used as a way of reformulating the input program in an equivalent but more efficient way.

4   EBG = PE

It has been the reformulation of EBG in terms of logical deduction (as given in [Kedar-Cabelli & McCarty, 1987]) which has made the definition of it precise enough in order to allow the comparison between EBG and PE. In order to see that the processes performed by EBG and PE as described above, although formulated in different contexts, are indeed the same when rephrased in terms of logic, we first identify the inputs of EBG with the inputs of PE.

- The goal concept of EBG corresponds to the name of the program to be partially evaluated.

- The domain-theory consists of the clauses and facts that constitute the program to be partially evaluated.

- The operationality criterion corresponds to the criterion that PE uses to stop the symbolic construction of the proof tree.

The normal PE algorithm is hardwired to only stop the construction of this proof tree at axioms in the form of single predicates (unit clauses, or facts, in Prolog terminology). However, as we shall see below, the PE algorithm can be trivially changed so that it becomes parametrised over the operationality definition.

- The training example of EBG corresponds to the values of the input variables $x_1, \ldots, x_k$.

However, as described above, PE does not really need the values for these variables. Furthermore, as remarked in [Mitchell *et al.*, 1986, page 67], EBG does not really need the training example either. In EBG the training example is only used to guide the algorithm through the relevant transformation of the goal concept, thereby reducing the search space for the explanation step, but it is not strictly necessary for the execution of this step. To facilitate the comparison of EBG and PE we can therefore assume for the moment that no training example is given, but we will return to this point in section 6. When no training example is given, the second step in the EBG algorithm becomes superfluous: it is no longer necessary to abstract away from the particular feature values of the training example.

Both algorithms now consist of only one step, namely constructing a proof tree for the goal concept (EBG terminology) or input predicate (PE terminology) using the rules of the domain theory, such that the leaves of this tree are all operational predicates. The conjunctions of all these leaves then forms the "reformulation of the goal concept" (EBG terminology) or the "more efficient program" (PE terminology).

It is now clear that both EBG and PE take as their inputs a goal predicate, a domain theory and an operationality criterion, and reformulate the original definition of the goal predicate in terms of operational predicates using the domain theory to justify the reformulation. Both EBG and PE perform this reformulation by constructing a proof tree with the goal predicate as the root-node and only operational predicates at the leaves, with the rules of the domain theory constituting the proof steps inside the tree.

5   EBG = PE by implementation

Figure 1 gives the definition of a small partial evaluator for Prolog programs [2]. [3]

---

[2]Throughout this paper we use the standard Edinburgh syntax for Prolog programs: variables start with uppercase letters or underscores, constants start with lowercase letters. As usual in Edinburgh syntax Prolog systems, the predicate `clause(Head, Body)` searches through the clause set for a clause that unifies with `Head:-Body` backtracking over clauses in the clause set for multiple solutions.

[3]This implementation of a partial evaluator only serves an illustrational purpose, and is barely of any practical value. To arrive at a practical partial evaluator, this program would have to be extended with facilities to deal with built-in predicates, and to avoid looping on recursive programs. However, the above program does illustrate the essential elements of the PE algorithm necessary for our discussion.

```
peval(Leaf, Leaf) :- clause(Leaf, true).
peval((Goal1, Goal2), (Leaves1, Leaves2)) :-
        peval(Goal1, Leaves1),
        peval(Goal2, Leaves2).
peval(Goal, Leaves) :-
        clause(Goal, Clause),
        peval(Clause, Leaves).
```

Figure 1: Partial evaluator.

The predicate `peval` takes a predicate as its input in the first argument and symbolically simulates the execution of this predicate, returning the leaves of the resulting proof tree in the second argument. Notice that the other input to PE (the code of the program to be optimised) is implicit in the Prolog database, using the `clause`-predicate). Not surprisingly, this partial evaluator looks very much like the standard Prolog interpreter that it simulates. Figure 2 shows such a Prolog interpreter in Prolog.

```
prolog(Leaf) :- clause(Leaf, true).
prolog((Goal1, Goal2)) :-
        prolog(Goal1),
        prolog(Goal2).
prolog(Goal) :-
        clause(Goal, Clause),
        prolog(Clause).
```

Figure 2: Prolog Interpreter.

The main difference between `peval` and `prolog` is that `peval` also returns the set of leaves of the proof tree as the result of the computation, thereby providing a reformulation of the input predicate in terms of the leaves of the tree.

The formulation of EBG shown in figure 3 is taken from [Kedar-Cabelli & McCarty, 1987, page 6] with small cosmetical changes (the 1st clause has been deleted, since it only served to remove vacuous `true` conjuncts from the final set of leaves, and conjunctions are no longer represented as lists, but rather using Prolog's standard ',',-functor).

```
ebg(Leaf, GenLeaf, GenLeaf) :-
        operational(Leaf), !,
        call(Leaf).
ebg((Goal1, Goal2), (GenGoal1, GenGoal2), (Leaves1, Leaves2)) :-
        ebg(Goal1, GenGoal1, Leaves1),
        ebg(Goal2, GenGoal2, Leaves2).
ebg(Goal, GenGoal, Leaves) :-
        clause(GenGoal, GenClause),
        copy((GenGoal:-GenClause), (Goal:-Clause)),
        ebg(Clause, GenClause, Leaves).
```

Figure 3: Explanation-based Generalisation.

The predicate `ebg` takes a training example as its first argument, and the goal concept as its second (thus, the first argument will be an instantiation of the second), and returns as its output in the third argument a list of conjuncts that form the

leaves of the proof tree for the goal predicate. Again, certain parts of the input to `ebg` are implicit: both the domain theory and the operationality criterion are provided from the Prolog database rather than being given as explicit arguments. As is clear from comparing figures 1 and 3, the EBG and PE code are very similar. They differ only in two small respects: the `ebg` code uses an explicit operationality criterion to stop the construction of the branches of the proof tree, whereas `peval` only stops when it finds unit clauses. The `peval` code can be trivially changed to take an explicit operationality criterion into account, by changing the first clause of figure 1 into

```
peval(Leaf, Leaf) :-
          operational(Leaf), !,
          call(Leaf).
```

and then supplying an appropriate definition of `operational`.

The second and more important difference between `peval` and `ebg` is the fact that `peval` includes the unification with operational predicates in the conjuncts it returns, thereby restricting the reformulation of the goal concept. `Ebg` on the other hand takes care not to include these unifications in the resulting set of operational leaves. The extra second argument of `ebg` is intended to perform exactly this role: the first and second argument of `ebg` maintain exactly the same proof tree, but for the unifications with operational predicates in clause 1, which are not included in the third argument, which contains the final set of leaves. Leaving out the unifications of operational predicates at the leaves of the proof tree corresponds to the generalisation step of the EBG algorithm. In the words of [Kedar-Cabelli & McCarty, 1987, page 5]:

> "The generalization is formed by propagating *rule substitutions* but ignoring *fact substitutions* when creating the proof tree."

However, the code for `peval` can be trivially changed to disregard unifications with operational predicates using the standard double-negation technique in Prolog to avoid variable bindings from successful goals. The updated version of clause 1 of the code for `peval` above has to be changed into

```
peval(Leaf, Leaf) :-
          operational(Leaf), !,
          not not call(Leaf).
```

A final difference between the code for `ebg` and `peval` to do with the generalisation process is the use of `copy` in the third clause of `ebg`. The role of the `copy` predicate is to ensure that the generalised tree exactly mirrors the specific tree, by having `GenGoal:-GenClause` and `Goal:-Clause` be exact copies, with new variables. However, as argued above, it is not really necessary to provide the EBG algorithm with a training example. Therefore we do not have to maintain two separate versions of the proof tree, one specific and one general version, but we only need the general proof tree, and we can remove the `copy` predicate from the code of `ebg`.

After these rather trivial changes, the remaining differences between the formulations of `peval` and `ebg` are purely cosmetic, showing that they are indeed the same algorithm.

6 The provision of a training example

As already mentioned above, EBG is usually described as needing a training example for the reformulation of the goal concept, but this training concept is not really necessary for the execution of EBG. The only purpose it serves is to restrict the number of possible reformulations of the goal concept by guiding the search through all possible rules from the domain theory. Although this reduction in the search space for the EBG algorithm is clearly desirable, there is a rather high price to be paid for this. The resulting reformulation of the goal concept is no longer equivalent to the original definition, but only represents sufficient (but possibly unnecessary) conditions for an example to be an instance of the goal concept. In [Mitchell *et al.*, 1986, page 57] this problem of sufficient-but-unnecessary conditions is mentioned as a topic for further research. However, as the comparison of EBG with PE shows, this problem can be solved by simply giving no training example at all. The S-M-N theorem then ensures us that the resulting reformulation will be equivalent to the original goal concept. Of course the computation is no longer guided by a target example, and is therefore much more expensive. Only when we are interested in just one special case is it still useful to provide a training example.

7 Worked example

We take the example from [Mitchell *et al.*, 1986] and [Prieditis & Mostow, 1987] to illustrate the behaviour of the code that has been developed above. This example is one that is often used in the machine learning literature, and comes originally from [Borgida *et al.*, 1985]. Consider the following Prolog database:

```
                  % Goal concept:
          safe_to_stack(X, Y) :-
                  lighter(X, Y).

                  % Example:
          on(box1, table1).
          volume(box1, 10).
          isa(box1, box).
          isa(table1, table).
          colour(box1, red).
          colour(table1, blue).
          density(box1, 10).

                  % Domain theory:
          lighter(X, Y) :-
                  weight(X, W1),
                  weight(Y, W2),
                  smaller(W1, W2).
          weight(X, 500) :- isa(X, table).
          weight(X, Y) :-
                  volume(X, V),
                  density(X, D),
                  times(V, D, Y).

                  % Operationality criterion:
          operational(Goal) :-
                  member(Goal, [times(_, _), smaller(_, _),
                                on(_, _), volume(_, _),
                                isa(_, _), colour(_, _),
                                density(_, _)]).
```

Figure 4: Stacking Example.

Figure 4 includes the full input for the EBG algorithm (and therefore for PE), excluding the definitions of the 'built-in' predicates `smaller` and `times`. Applying both `peval` and `ebg` to this example, by calling

```
        ?- peval(safe_to_stack(X, Y), Reformulation).
```

and

```
      ?- ebg(safe_to_stack(box1, table1), safe_to_stack(X, Y),
                      Reformulation).
```

gives the result:

```
        Reformulation = (volume(X, VX), density(X, DX),
                         times(VX, DX, MX), isa(Y, table),
                         smaller(MX, 500))
```
(i.e. anything lighter than 500 can be stacked on a table).

Notice that in this example, the `ebg` predicate was guided by its input example, whereas the `peval` performed an unguided search. As a result, `ebg` only computes the above reformulation of the goal criterion, whereas the unguided computation performed by `peval` also returns the following two reformulations:

```
Reformulation = (isa(X, table), volume(Y, VY), density(Y, DY),
                 times(VY, DY, MY), smaller(500, MY))
```
(i.e. a table can be stacked on anything heavier than 500)

```
Reformulation = (volume(X, VX), density(X, DX), times(VX, DX, MX),
                 volume(Y, VY), density(Y, DY),
                 times(VY, DY, MY), smaller(MX, MY))
```
(i.e. anything can be stacked on anything lighter)

These three formulations are generated by the two definitions of `weight` in the domain theory. Since this definition is used twice (in the predicate `lighter`), we would potentially get four reformulations of the goal concept, but the fourth formulation would involve stacking two `table`s, both with weight 500, making the operational predicate `smaller` false. The fourth reformulation is therefore rejected. In this small example it is easy to see that the disjunction of all four alternative reformulations forms exactly the set of leaves of the fully expanded proof tree for a proof of the goal concept. This shows that PE gives a full equivalent reformulation of the goal concept (namely the disjunction of all four alternative reformulations), while EBG (guided by its example), only gives a reformulation whose conditions can in general be too strong.

8   Conclusions

After having given a general description of explanation based generalisation and partial evaluation, we have used a specific formulation of both of them in terms of logical deduction to show that the two algorithms are essentially equivalent, modulo a few minor differences to do with the stop criterion and with the way the two formalisms treat unifications arising from leaves in the proof tree.

In general we believe that such rational reconstructions of apparently unrelated algorithms in terms of each other is a useful activity, not only because it prevents reinventing the wheel, but also because often such rational reconstructions generate new insights in and additions to both versions of the algorithm. In this case, both EBG and PE benefit from the comparison:

- It becomes clear that EBG consists strictly of a deductive reformulation of the original goal concept in terms of operational concepts.

- The comparison with PE shows us that there is no reason why the training example of EBG should be a ground term, although this is suggested by all the examples in the papers on EBG referred to above.

- In fact, as argued in section 6 above, a training example is not necessary at all for EBG. This means that a balance can be struck between the cost of the search during the learning algorithm (high when no training example provided) and the generality of the concept that is learned (more general when no training example is provided).

- There is no strict need for PE to unfold the full proof tree all the way to the final leaves, but it is possible to stop this process at previously specified predicates, as defined by an explicit operationality criterion. This insight is important because it might be used in solving two of the major problems associated with PE, namely firstly the termination of unfolding recursively defined predicates (it is not clear when to stop the potentially infinite unfolding

of these predicates), and secondly the increase in size of the code after partial evaluation (this often exponential increase is due to the unfolding of all the conditional branches in the code).

- The current work on extending EBG to deal with imperfect theories ([Rajamoney & DeJong, 1987] and [Doyle, 1986]) might have interesting implications for the logic programming community.

- The same holds for the work of Utgoff and others [Utgoff & Mitchell, 1982], showing that EBG leads to inventing "new" predicates, that were not explicit in the original domain theory.

- A lot of current work in the PE field, in particular on the possibilities of rejoining conditional branches and on heuristic guidance to the PE algorithm will be of similar interest to workers in explanation-based machine learning.

Note

After this paper had been provisionally accepted for publication, it was brought to our attention that similar results have been achieved simultaneously and independently in [Prieditis, 1988].

References

[Borgida *et al.*, 1985]   A. Borgida, T. M. Mitchell, and K. E. Williamson. Learning improved integrity constraints and schemas from exceptions in data and knowledge bases. In M. L. Brodie and J. Mylopoulos, editors, *On knowledge base management systems*, New York, 1985. Springer Verlag.

[DeJong, 1981]   G. DeJong. Generalizations based on explanations. In A. Drinan, editor, *Proceedings of the Seventh IJCAI*, pages 67–69, Vancouver, Canada, August 1981. International Joint Conference on Artificial Intelligence.

[Doyle, 1986]   R. Doyle. Constructing and refining causal explanations from an inconsistent domain theory. In *Proceedings of AAAI-86*, pages 538–544, Philadelphia, PA, 1986. American Association for Artificial Intelligence, Morgan Kaufmann.

[Ershov, 1982]   A. P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[Futamura, 1971]   Y. Futamura. Partial evaluation of computation process- an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[Kedar-Cabelli & McCarty, 1987]   S. Kedar-Cabelli and L. T. McCarty. Explanation-based generalization as resolution theorem proving. In P. Langley, editor, *Proceedings of the 4th International Machine*

	Learning Workshop, pages 383–389, University of California, Irvine, CA, 1987. Morgan Kaufmann.
[Kleene, 1962]	S. Kleene. *Introduction to Metamathematics.* North Holland, Amsterdam, 1962.
[Komorowski, 1982]	H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of PROLOG. In *Proceedings of the ninth conference on the Principles of Programming Languages (POPL)*, pages 225–267, Albuquerque, New Mexico, 1982. ACM.
[Mitchell, 1982]	T. M. Mitchell. Toward combining empirical and analytical methods for inferring heuristics. Technical Report LCSR-TR-27, Laboratory for Computer Science Research, Rutgers University, 1982.
[Mitchell, 1983]	T. M. Mitchell. Learning and problem solving. In Alan Bundy, editor, *Proceedings of the Eighth IJCAI*, pages 1139–1151, Karlsruhe, Germany, 1983. International Joint Conference on Artificial Intelligence.
[Mitchell *et al.*, 1986]	T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986. Also available as Tech. Report ML-TR-2, SUNJ Rutgers, 1985.
[Nilsson, 1980]	N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Pub. Co., Palo Alto, California, 1980.
[Prieditis & Mostow, 1987]	A. E. Prieditis and J. Mostow. PROLEARN: Towards a Prolog interpreter that learns. In K. Forbus and H. Shrobe, editors, *Proceedings of AAAI-87*, pages 494–498, Seattle, Washington, July 1987. Morgan Kaufmann. Also available as DAI Research Report No. ML-TR-13, Lab. for Computer Science Research, Hill Center for Mathematical Sciences, Rutgers University, New Brunswick, New Jersey.
[Prieditis, 1988]	A. E. Prieditis. Environment-guided program transformation. In *Proceedings of the AAAI Spring Symposium*, Stanford, CA, March 1988. American Association for Artificial Intelligence.
[Rajamoney & DeJong, 1987]	S. Rajamoney and G. DeJong. The classification, detection and handling of imperfect theory problems. In J. McDermott, editor, *Proceedings of the Tenth IJCAI*, pages 205–207, Milan, Italy, August 1987.

[Takeuchi & Furukawa, 1986]   A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H. J. Kugler, editor, *Proceedings of IFIPS '86*, pages 415–420, Dublin, 1986. IEEE. also: ICOT Research Centre, Technical Report TR-125, Tokyo, July 1985.

[Utgoff & Mitchell, 1982]   P. E. Utgoff and T. M. Mitchell. Acquisition of appropriate bias for inductive concept learning. In *Proceedings of AAAI-82*, pages 414–417, Pittsburgh, August 1982. American Association for Artificial Intelligence.

[Venken, 1984]   R. Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation. In *Proceedings of the 6th European Conference on Artificial Intelligence*, pages 91–100, Pisa, Italy, September 1984. ECAI.

[Waldinger, 1977]   R. Waldinger. *Achieving Several Goals Simultaneously*, volume 8 of *Machine Intelligence*, chapter 6, pages 94–138. Halstead and Wiley, New York, 1977.