Imperial College London
Department of Computing

Synthesis of Functional Programs using Answer Set Programming

James Thomas Rodden

Contents

Chapter 1

Background

1.1 Answer Set Programming

1.1.1 The Stable Model Semantics

The Stable Model Semantics provides a natural semantics for both normal and extended logic programs, in which instead of giving individual solutions to queries we define a set of "Answer Sets" (Stable Models) of the program. For example, given the program:

$$p \leftarrow \mathsf{not}\ q.$$

$$q \leftarrow \mathsf{not}\ p.$$

Because the value of p depends on the value of q, when you query Prolog for p? this program results in an infinite search, alternately searching for p and q until cancelled. However, using the Stable Model Semantics, this particular program has an answer set for each solution : {p} and {q}.

Grounding

Before you can solve an extended logic program using Answer Sets you must first ground it. To calculate the grounding of a program P you replace each rule in P by every ground instance of

that rule. For example, program

$$p(1,2).$$
$$q(X) \leftarrow p(X,Y),\ \mathsf{not}\ q(Y).$$

has grounding

$$p(1,2).$$
$$q(1) \leftarrow p(1,1),\ \mathsf{not}\ q(1).$$
$$q(1) \leftarrow p(1,2),\ \mathsf{not}\ q(2).$$
$$q(2) \leftarrow p(2,1),\ \mathsf{not}\ q(1).$$
$$q(2) \leftarrow p(2,2),\ \mathsf{not}\ q(2).$$

Typically, a program containing functions has an infinite grounding. For example,

$$q(0, f(0)).$$
$$p(X) \leftarrow q(X,Y),\ \mathsf{not}\ p(Y).$$

This program would have an infinite grounding as there are an infinite amount of atoms 0, f(0), f(f(0)).. and so on. ASP solvers deal with this issue by incrementally calculating the grounding, by only generating rules such that for each atom A in the positive literals of the rule there exists another rule A as the head. This would give program 1.10 the grounding :

$$q(0, f(0)).$$
$$p(0) \leftarrow q(0, f(0)),\ \mathsf{not}\ p(f(0)).$$

Safety

Because of the need for ASP solvers to ground the entire program, they are restricted to safe rules. A rule is safe if every variable in that rule occurs positively in the body of the rule. Some examples of unsafe rules are:

$$p(X) \leftarrow q(Y).$$
$$p(Z) \leftarrow \text{not } q(X,Y),\ r(X,Y).$$
$$p(X) \leftarrow q(X),\ \text{not } r(Y).$$

Least Herbrand Models

Each normal logic program P has a Herbrand Base, the set of all ground atoms of the program. The program can have a Herbrand Interpretation which assigns each atom in its base a value of true or false, typically written as the set of all atoms which have been assigned true. Then, a Herbrand Model M of P is a Herbrand Interpretation where if a rule in P has its body satisfied by M then its head must also be satisfied by M.

A Herbrand Model M is minimal if no subset of M is also a model. For definite logic programs this model is unique and called the least Herbrand Model, however this does not always hold for normal or extended logic programs.

For example, the program

$$p \leftarrow \text{not } q.$$
$$q \leftarrow \text{not } p.$$

has two minimal models, {p} and {q}.

1.1.2 Calculating Answer Sets

Reduct

Let P be a ground normal logic program, and X be a set of atoms. The reduct of P, P^X is calculated from P by :

1. Delete any rule from P which contains the negation as failure of an atom in X.
2. Delete any negation as failure atoms from the remaining rules in P.

We then say X is an Answer Set (Stable Model) of P if it is the least Herbrand Model of P^X. For example, if program P is:

$$p(1,2).$$
$$q(1) \leftarrow p(1,1),\ \mathsf{not}\ q(1).$$
$$q(1) \leftarrow p(1,2),\ \mathsf{not}\ q(2).$$
$$q(2) \leftarrow p(2,1),\ \mathsf{not}\ q(1).$$
$$q(2) \leftarrow p(2,2),\ \mathsf{not}\ q(2).$$

Then, when trying X = {p(1,2), q(1)}, the reduct is:

$$p(1,2).$$
$$q(1) \leftarrow p(1,2).$$
$$q(2) \leftarrow p(2,2).$$

The Least Herbrand Model $M(P^X)$ = {p(1,2), q(1)}, so X is an Answer Set of P.

It is important to note that Answer Sets are not unique and some programs might have no Answer Sets.

Constraints

Constraints are a way of filtering out unwanted Answer Sets. They are written as rules with an empty head, which is equivalent to $\bot$. Semantically, this means that any model which satisfies the body of the constraint cannot be an Answer Set. For example, the program with constraint

$$p \leftarrow \mathsf{not}\ q.$$

$$q \leftarrow \mathsf{not}\ p.$$

$$\leftarrow p,\ \mathsf{not}\ q.$$

Has only one Answer Set, {q}.

Aggregates and Choice Rules

An aggregate is a ASP atom with the format $a\ \mathsf{op}\ [h_1 = w_1, h_2 = w_2, \ldots, h_n = w_n]\ b$. An aggregate is satisfied by an interpretation X if operation op applied to the multiset of weights of true literals is between the upper and lower bounds a and b.

I will be mainly using one application of aggregates, the choice rule. A choice rule has an aggregate using the count operation as the head of the rule, and semantically represents a choice of a number of head atoms, between the upper and lower bound. For example, the choice rule $1\ \{value(Coin, heads), value(Coin, tails)\}\ 1 \leftarrow coin(Coin)$. represents that a coin can either have value heads or tails but not both.

To calculate the Answer Sets of a program P which contains aggregates by adding an additional step to the construction of the reduct. For each rule with an aggregate:

1. If the aggregate is not satisfied by X then convert the rule into a constraint by removing the aggregate, replacing it with
2. If the aggregate is satisfied by X then generate one rule for each atom A in the aggregate which is also in X, with A as the head.

For example, consider program P with X = {p ,q, r}:

$$1\ \{p, q\}\ 2 \leftarrow r.$$

$$r.$$

the reduct is then

$$p \leftarrow r.$$

$$q \leftarrow r.$$

$$r.$$

Optimisation Statements

1.1.3 Learning from Answer Sets

Brave and Cautious Induction

ASPAL

1.1.4 ASP Solvers

CLASP

1.2 Inductive Functional Programming

- Inductive Functional Programming is the automatic synthesis of program
- Applications
- Traditionally, there have been two approaches to IFP. The analytical approach performs pattern matching on the given examples, usually performing a two-step process of first generalising the examples and then folding this generalisation into a recursive program.

The "generate and test" search approach works by generating an infinite stream of candidate programs and then test these candidates to see if they correctly model the examples.

Each approach has its own advantages and disadvantages. The analytical approach, while fast, can be very limited in its target language and the types of programs it can generate, typically being limited to reasoning about data structures such as lists or trees. On the other hand, the search based approach is a lot less restricted but has worse performance due to the potentially huge search space. This issue is minimised by various optimisations performed on the generated programs, typically using a subset of the examples as an initial starting point.

- Overview of current approaches and how they relate

1.3 Haskell