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:

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

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

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

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:

$$
\begin{aligned}
&p(1,2).\\
&q(1) \leftarrow p(1,1),\ \mathsf{not}\ q(1).\\
&q(1) \leftarrow p(1,2),\ \mathsf{not}\ q(2).\\
&q(2) \leftarrow p(2,1),\ \mathsf{not}\ q(1).\\
&q(2) \leftarrow p(2,2),\ \mathsf{not}\ q(2).
\end{aligned}
$$

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

$$
\begin{aligned}
&p(1,2).\\
&q(1) \leftarrow p(1,2).\\
&q(2) \leftarrow p(2,2).
\end{aligned}
$$

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 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 $\bot$.

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

An Optimisation Statement is an ASP atom of the form $\#op\,[l_1 = w_1, l_2 = w_2, \ldots, l_n = w_n]$, where $\#op$ is either $\#maximise$ or $\#minimise$, and each l_n is a ground literal assigned a weight w_n. They allow for an ASP solver to search for optimal Answer Sets which maximise or minimise the sum of the atoms with regards to their weights. For instance,

$$p \leftarrow \mathsf{not}\ q.$$
$$q \leftarrow \mathsf{not}\ p.$$
$$\#minimise\,[p = 1, q = 2]$$

Has one optimal answer set, {p}.

1.1.3 Learning from Answer Sets

I will first introduce the idea of Brave and Cautious Entailment:

- We say an atom A is Bravely entailed by a program P if it is true in at least one Answer

Set of P.

- We say an atom A is Cautiously entailed by a program P if it is true in all Answer Sets of P.

Cautious Induction

A Cautious Induction task is defined as a search for Hypothesis H, given a background knowledge B (an ASP program) and sets of positive and negative examples (atoms) E^+ and E^-.

We want to find H such that:

- $B \cup H$ has at least one Answer Set
- For all Answer Sets A of H :
  - $\forall e \in E^+ \, : \, e \in A$
  - $\forall e \in E^- \, : \, e \notin A$

Brave Induction

Brave Induction is defined similarly to Cautious Induction. Given a background knowledge B and sets of examples E^+ and E^-, we want to find a Hypothesis H such that, there is at least one Answer Set A of H that:

- $\forall e \in E^+ \, : \, e \in A$
- $\forall e \in E^- \, : \, e \notin A$

For example, consider Program P:

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

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

$$s \leftarrow r.$$

with $E^+ = \{\text{s}\}$ and $E^- = \{\text{q}\}$.

The Hypothesis H = {p, s} is both a Brave and Cautious Inductive Solution, as there is only one Answer Set, and it satisfies all of the examples.

The Hypothesis H = {s} is a Brave Inductive Solution but not a Cautious one. It has two Answer Sets, {p, s} and {q, s}, but only {p, s} satisfies the examples.

ASPAL

ASPAL is an ASP algorithm which computes the solution of a Brave Inductive task by encoding it as an ASP program which has the solutions as the Answer Sets of the program.

First, we encode the Hypothesis space by generating a number of skeleton rules. Each skeleton rule represents a possible rule in the Hypothesis (with constant terms replaced with variables), together with an identifier. The atom rule(ID, $c_1, \ldots, c_n$) represents the choice of skeleton rule with ID and the constant variables replaced by various c_n. The goal of the task is to find these rule atoms.

We next rule out Answer Sets which do not fit the examples by adding a goal rule and a constraint, of the format :

$$\mathsf{goal} \leftarrow e_1^+, \ldots, e_n^+,\ \mathsf{not}\ e_1^-, \ldots,\ \mathsf{not}\ e_n^-$$
$$\leftarrow \mathsf{not\ goal}$$

This rules out any answer set which does not contain all of the positive examples and contains any negative examples.

We then make sure the Answer Sets of the program correspond to the optimal solutions of the task by adding an optimisation statement

$$\#minimise\,[rule(ID, c_1, \ldots, c_n) = ruleLength, \ldots].$$

where there is a rule predicate for each skeleton rule, weighted by the length of that rule.

As an example of the entire ASPAL encoding, consider the program :

$$\begin{aligned} bird(a). &\qquad bird(b).\\ ability(fly). &\qquad ability(swim).\\ can(a, fly). &\qquad can(b, swim). \end{aligned}$$

with mode declarations

$$\begin{aligned} &modeh(penguin(+bird)).\\ &modeb(\mathsf{not}\ can(+bird, \#ability)). \end{aligned}$$

and examples $E^+ = \{\text{penguin(b)}\}$ and $E^- = \{\text{penguin(a)}\}$.
This task has encoding:

$penguin(V1) \leftarrow bird(V1),\ rule(1).$

$penguin(V1) \leftarrow bird(V1),\ \textsf{not}\ can(V1, C1), rule(2, C1).$

$penguin(V1) \leftarrow bird(V1),\ \textsf{not}\ can(V1, C1),\ \textsf{not}\ can(V1, C2), rule(3, C1, C2).$

%%%%%

$\{rule(1), rule(2, fly), rule(2, swim), rule(3, fly, swim)\}.$

$\#minimise\ [rule(1) = 1, rule(2, fly) = 2, rule(2, swim) = 2, rule(3, fly, swim) = 3].$

%%%%%

$goal \leftarrow pengiun(b),\ \textsf{not}\ penguin(a).$

$\leftarrow\ \textsf{not}\ goal.$

1.1.4 ASP Solvers

Typically, ASP solvers work in two parts.

- First the input program must be converted into the ground finite logic program by a grounder. Typically this step also includes optimisations to help the solver.
- The ground program is then passed into the solver which calculates the Answer Sets of the program.

The main popular ASP solvers are SMODELS, DLV and CLASP. CLASP is the ASP solver which I will be using to run my learning tasks as part of this project. It consists of three programs : The grounder *gringo*, the ASP solver *clasp* and the utility *clingo* which combines the two.

Gringo

Gringo calculates the ground logic program by replacing the variables in the program by ground terms. The resulting program needs only be an equivalent program (meaning it has the same Answer Sets), to be able to deal with programs with infinite number of Answer Sets.

1.2 Inductive Functional Programming

Inductive Functional Programming is the automatic synthesis of declarative programs from an incomplete specification, typically given as pairs of input-output examples.

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.

1.2.1 Conditional Constructor Systems

Programs are represented in a functional style as a *term rewriting system*, i.e a set of rules of the form $l \to r$. The LHS of the rule, l, has the format $F(a_1, \ldots, a_n)$ where F is a user-defined function and a_1 to a_n are variables or pre-defined data type functions(constructors). In addition, rules must be *bound*, meaning that all variables that appear on the right hand side of a rule must also appear on the left hand side.

We can also extend these rules with conditionals, which must be met if the given rule is to be applied. Conditionals are of the form $l \to r \Leftarrow v_1 = u_1, \ldots, v_n = u_n$, where each $v_n = u_n$ is an *equality constraint* which has to hold for the rule to be applied.

EXAMPLE TRS HERE.

Using this definition we can begin to learn TRS. We take the sets of positive and negative examples (E^+ and E^-), written as sets of input/output examples in the form of unconditional re-write rules, and a *background knowledge* BK which is a set of additional re-write used to help computation.

The goal of the learning task is then to a finite set of rewrite rules R such that:

- $R \cup BK \models E^+$ (covers all of the positive examples).
- $R \cup BK \not\models E^-$ (covers none of the negative examples).

Because the target domain of this task is often very large, two extra constraints are added to the type of rule we are allowed to learn.
Restriction Bias is similar to the language bias typically found in ILP systems, and restricts the format of the learned rules and conditionals, for example to avoid mutual recursion.
Preference Bias introduces an idea of optimisation to the task such that the learned rules are optimal while also satisfying the examples. Some such optimisations are preference to shorter right hand sides or preference to as few rules as possible.

1.2.2 Overview of current tools

Here I will discuss two modern IFP systems which I will use to compare to my system as part of my evaluation:

- **MagicHaskeller** is based on the generate and test approach to IFP, and works by generating a stream of progressively more complicated programs and then testing them against the given specification. This search is exhaustive and performed in a breadth-first manner with application of a variant of Spivey's Monad.
  MagicHaskeller has the advantage of being easy to use as you only have to give the specification of the desired function as an argument, which is typically no more than a few lines of code in the target language. The tool also has a particularly large component library of built-in Haskell primitives which the tool uses to s
- **Igor II**

1.3 Haskell?

Chapter 2

Method

2.1 A Haskell Interpreter in ASP

As a way of better understanding how to represent the target language of the tool, my first step was to implement an Interpreter for the simple target language of my tool.

2.1.1 Target Language

Initially, I chose to target a simple sub language of Haskell with the following terms :

- Addition
- Subtraction
- Multiplication
- Function Calls (including recursion) with any number of arguments.
- Lists

I chose these terms as I felt they were expressive enough to be able to represent sufficiently complicated test examples.

I then represent each line of the target Haskell program with a Rule with a rule(Num, FuncName, Input, Output) predicate in the head. For example, we can represent the recursive factorial Haskell program

```
f x :: Int -> Int

f 0 = 1

f x = x * f(x-1)
```

Is represented by the ASP program

```
rule(1, f, 0, 1) :- input(call(f, 0)).

rule(2, f, N, mul(N, call(f, sub(N,1)))) :- input(call(f, N)).
```

We add the input atom to the body of the rule to constrain the grounding of these rules.

2.1.2 Evaluating Rules

The goal of my interpreter is to be able to evaluate these rules with some given input, to get the correct output. Because we are interpreting Functional Programs we can be sure each input has one output.

```
output(call(F, Args), Out) :- rule(Index, F, Args, Expr), match(F, Index,
    Args), value(Expr, Out).
```

This rule generates output predicates if there is a rule that best matches the given arguments, and the output of that rule has a known value.

```
value(mul(A, B), V1 * V2) :- eval(mul(A, B)), value(A, V1), value(B, V2).

value(sub(A, B), V1 - V2) :- eval(sub(A, B)), value(A, V1), value(B, V2).

value(add(A, B), V1 + V2) :- eval(add(A, B)), value(A, V1), value(B, V2).

value(call(F, Args), Out) :- eval(call(F, Args)), output(call(F, Inputs),
    Out), value(Args, Inputs).

value((A, B), (V1, V2)) :- eval((A, B)), value(A, V1), value(B, V2).

value(N, N) :- eval(N), const(N).
```

These rules handle calculating the value of expressions. The eval() predicate is generated whenever you need to calculate the value of an expression.

```
eval(Expr) :- input(call(F, Inputs)), rule(Index, F, Inputs, Expr), match(F,
    Index, Inputs).
eval(Args) :- eval(call(F, Args)).
eval(A) :- eval((A, B)).
eval(B) :- eval((A, B)).
eval(A) :- eval(mul(A, B)).
eval(B) :- eval(mul(A, B)).
eval(A) :- eval(sub(A, B)).
eval(B) :- eval(sub(A, B)).
eval(A) :- eval(add(A, B)).
eval(B) :- eval(add(A, B)).
```

These rules handle which expressions to calculate values for. The most important rule here is to top one, which says that whenever you have an input to some matching rule then you need to evaluate the output of that rule.

```
input(call(F, Inputs)) :- eval(call(F, Args)), value(Args, Inputs).
```

This rule generates more input predicated if we have to calculate the value for a call to a function. This allows us to then generate more ground rule predicates and repeat.

```
match(F, Index, Inputs) :- not smaller_match(F, Index, Inputs), rule(Index,
    F, Inputs, _), match2(F, Index, Inputs).
smaller_match(F, Index, Args) :- match(F, 0, Args), 0 < Index, const(Index).
```

These rules allow us to match to the most fitting rule. A rule matches some arguments if there is no smaller rule that also matches those arguments. The match2() predicate is there as a placeholder, and will have effect when more complicated matching (e.g guards) will be needed.

2.2 Initial Learning

I decided to base my (initial) learning task as an ASP encoding similar to ASPAL, with the goal to learn the correct rule() predicates given a number of input/output example pairs:

- My background knowledge is simply my interpreter from 2.1.2

- I characterised the Skeleton Rules as a choice rule where there can be up to a given number of rule predicates with an expression that is structured correctly.

```
0 {rule(RuleNum, FuncName, Args, Expr) : RuleNum = 1..N :
   valid_args(Args) : valid_expr(Args, Expr)} N :- fn_name(FuncName, N).
```

- I then add a constraint to remove any answer sets which do not produce the correct output against the given examples.

```
:- not output(In, Out), example(In, Out).
```

2.3 Future Work

- Before the end of this term (11th March) I hope to have a basic learning system in place where I can learn programs with an arbitrary number of arguments, including lists.

- My goals after Easter will be primarily focused on extending the language bias to deal with guards, let / where statements and higher order functions. At the same time I would like to attempt to apply existing IFP optimisations to my implementation and run experiments to compare my new tool to the most up to date IFP algorithms.