Imperial College London
Department of Computing

Synthesis of Functional Programs using Answer Set Programming

James Thomas Rodden

Contents

Chapter 1

Introduction

Inductive Functional Programming (IFP) is the automatic synthesis of declarative programs from an incomplete specification, typically given as pairs of Input-Output examples. Whilst this field has existed since the 1970s, recent work has slowed due to limitations on the complexity and structure of programs that can be learned.

My project introduces a new approach to IFP, through the use of Answer Set Programming (ASP). Last year there was a successful project to inductively compute imperative programs using ASP so this project aims to apply a similar technique to functional programs, with the difference being that functional programming is focused more approaches like recursion than the imperative approach.

ASP is a relatively new approach to logic programming, which works by computing the so called "Answer Sets" of a logic program which correspond the minimal models of that program. ASP has the advantage over traditional logic programming languages (i.e Prolog) that it is much more expressive, which gives me a lot of flexibility in my solution.

Chapter 2

Background

2.1 Answer Set Programming

2.1.1 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 :- not q.
q :- not p.
```

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) :- p(X, Y), not q(Y).
```

has grounding

```
p(1,2).
q(1) :- p(1, 1), not q(1).
q(1) :- p(1, 2), not q(2).
q(2) :- p(2, 1), not q(1).
q(2) :- p(2, 2), not q(2).
```

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

```
q(0, f(0)).
p(X) :- q(X, Y), 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) :- q(0, f(0)), 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) :- q(Y).
p(Z) :- not q(X, Y), r(X, Y).
p(X) :- q(X), 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 :- not q.
q :- not p.
```

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

2.1.2 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 a least Herbrand Model of P^X. For example, if program P is:

```
p(1, 2).
q(1) :- p(1, 1), not q(1).
q(1) :- p(1, 2), not q(2).
q(2) :- p(2, 1), not q(1).
q(2) :- p(2, 2), not q(2).
```

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

```
p(1, 2).
q(1) :- p(1, 2).
q(2) :- p(2, 2).
```

A 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 :- not q.
q :- not p.
:- p, 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 $\bot$.
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 :- r.
r.
```

the reduct is then

```
p :- r.
q :- 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 :- not q.
q :- not p.
#minimise [p = 1, q = 2].
```

Has one optimal answer set, {p}.

2.1.3 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 the program:

```
p :- not q.
q :- not p, not r.
s :- r.
```

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

`goal :-` $e_1^+, \ldots, e_n^+,$ `not` $e_1^-, \ldots,$ `not` e_n^-
`:- 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, ...].`

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 :

```
bird(a).  bird(b).
ability(fly).  ability(swim).
can(a, fly).  can(b, swim).
```

with mode declarations

```
modeh(penguin(+bird)).
modeb(not can(+bird, #ability)).
```

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

```
penguin(V1) :- bird(V1), rule(1).
penguin(V1) :- bird(V1), not can(V1, C1), rule(2, C1).
penguin(V1) :- bird(V1), not can(V1, C1), 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 :- penguin(b), not penguin(a).
:- not goal.
```

2.1.4 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.

2.2 Inductive Functional Programming

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.

2.2.1 Conditional Constructor Systems

Programs are represented in a functional style as a *term rewriting system*, i.e a set of rules of the form $l \rightarrow r$. 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.

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.

2.2.2 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 where possible.

  However, MagicHaskeller does not have the ability to compute large programs, due to the performance cost of exhaustive breadth-first search. This problem, however, is typical of the generate and test approach.

- **Igor II** is based on the analytical approach is specialised towards learning recursive programs. It works by first calculating the *least-general generalisation* of the examples, which extracts terms which are common between the examples and terms which are uncommon are represented as variables. If this initial hypothesis is incomplete (i.e not a correct functional program) then four refinement operators are applied to attempt to complete it:

1. Partition the examples into two subsets and generate a new, more specific least general generalisation for each subset. This partition is calculated using pattern matching on the variables from the lhs of the initial hypothesis.
2. If the rhs of the initial hypothesis has a function as its root element then each unbound argument of this function is treated as a subproblem, and new functions are introduced for each argument.
3. The rhs of the initial hypothesis may be replaced by a recursive call to a user-defined function, where the arguments of this call are newly introduced functions.
4. If the examples match certain properties then a higher order function may be introduced, and synthesising the arguments to this function becomes the new induction problem.

Igor II has a number of limitations. If there are a large number of examples then performance can be impacted because of the large number of combinations when matching to a defined function. Synthesis can fail or perform incorrectly if some examples from the middle of the set are removed, meaning if we want to specify a particularly large or complex example we also have to specify all of the smaller examples, reducing performance.

References

[1] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. *A User's guide to gringo, clasp, clingo and iclingo*, 2010.

[2] M. Hofmann. *Igor II - an Analytical Inductive Functional Programming System*, 2010.

[3] M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *The Second Conference on Artificial General Intelligence*, 2009.

[4] S. Katayama. An analytical inductive functional programming system that avoids unintended programs. In *PEPM '12 Proceedings of the ACM SIGPLAN 2012 Workshop on Partial evaluation and program manipulation*, pages 43–52, 2012.

[5] M. Law. Notes on ASP from course 304, Logic Based Learning.

[6] M. Sergot. Notes on Answer Sets and The Stable Model Semantics from course 491, Knowledge Representation.

Chapter 3

The Initial Approach : A Haskell Interpreter in ASP

3.1 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.

3.1.1 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.

3.1.2 Program Representation

To represent a generic Haskell program in ASP, first I had to choose various predicates to represent different parts of the program.

For the arithmetic operations present in my target language, I decided to use simple binary predicates. For example, addition is represented by the `add(X, Y).` predicate, and subtraction and multiplication follow in the same way.

To represent function calls, I have introduced the `call(F, Args).` predicate. Here, F is the name of the function being called, and Args represents an (arbitrary length) list of function

arguments.

Using these predicates, I can represent the Haskell implementation of the recursive factorial program

```
f x :: Int -> Int
f 0 = 1
f x = x * f(x-1)
```

by the ASP program

```
rule(1, f, 0, 1) :- input(call(f, 0)).
rule(2, f, N, x1) :- input(call(f, N)).

where(x1, N, mul(N, x2)) :- input(call(f, N)).
where(x2, N, call(f, x3)) :- input(call(f, N)).
where(x3, N, sub(N, 1)) :- input(call(f, N)).
```

To explain further, each line (or recursive case) of the target Haskell program is represented by with a rule with a `rule(Num, FuncName, Input, Output).` predicate in the head.

Because of the vast algorithmic complexity of the combinations of arithmetic operations, I was struggling with performance issues. To sufficiently cover the possible program space, the required number of skeleton rules was overwhelming even for comparatively simple programs.

As a solution to this issue, I make heavy use of `where(Var, Args, Body).` predicates. These "where" predicates are semantically identical to the Haskell equivalent, specifying replacement rules for the given variables in the scope of the rule.

For example, in the program above, instead of one rule

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

We instead use

```
rule(2, f, N, x1) :- input(call(f, N)).

where(x1, N, mul(N, x2)) :- input(call(f, N)).
where(x2, N, call(f, x3)) :- input(call(f, N)).
where(x3, N, sub(N, 1)) :- input(call(f, N)).
```

nesting each operation in its own clause.

3.1.3 Evaluating Rules

The goal of my interpreter is to be able to evaluate these rules with some given input, to get the correct output.

Internal Predicates

As part of the interpreter, I have declared a number of my own predicates, some of which it may not be clear what they represent. As such, this section details them in a succinct manner.

- `input(Expr)` and `output(Expr, Val)` : These predicates represent input and output to the learned function. If a function has some input Expr, then it is expected that there is an output predicate with a matching Expr.
- `match(F, Index, Inputs)` : This predicate represents which function inputs match a respective line in the program. It may be helpful to think of this predicate as a concrete way of stating pattern matching behaviour.
- `value(Expr, Val)` : This predicate represents the value of a given simple expression, i.e `sub(3, 1)` has value 2.
- `value_with(Expr, Val, Args)` : This predicate represents the value of an expression, given input arguments. The extra argument is necessary due to the inclusion of "where" predicates. As the expression "x1" could have multiple values throughout execution, I have to keep track of the value at each step of execution.
- `eval_with(Expr, Args)` : This predicate is used to represent which expressions we want to find the value of, because we do not want to find the value of every possible expression, only expressions we care about. As with `value_with`, the extra argument is necessary as the values of "where" clauses could change throughout execution.
- `complex(Expr)` and `n_complex(Expr)` : A predicate is "complex" if it references a "where" variable. This predicate is used to determine if the `value_with` or `value` predicate should be used.
- `check_if_complex(Expr)` : Similarly to `eval_with`, because we don't want to check if every possible expression is complex, just ones we care about.

Computation Rules

To generate the above predicates, I needed to specify the relations between them. These generation (or computation) rules are detailed below.

To start, I needed a rule to calculate the output, given inputs and a rule to follow.

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

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_with(mul(A, B), @to_num(V1) * @to_num(V2), Args) :- eval_with(mul(A,
   B), Args), value_with(A, V1, Args), value_with(B, V2, Args),
   const_number(@to_num(V1)*@to_num(V2)).
value_with(sub(A, B), @to_num(V1) - @to_num(V2), Args) :- eval_with(sub(A,
   B), Args), value_with(A, V1, Args), value_with(B, V2, Args),
   const_number(@to_num(V1)-@to_num(V2)).
value_with(add(A, B), @to_num(V1) + @to_num(V2), Args) :- eval_with(add(A,
   B), Args), value_with(A, V1, Args), value_with(B, V2, Args),
   const_number(@to_num(V1)+@to_num(V2)).
```

These rules handle calculating the value of arithmetic expressions. Note the use of a lua function `@to_num()`, as a bug workaround.

```
value_with(call(F, Args_new), Out, Args_old) :- eval_with(call(F, Args_new),
   Args_old), output(call(F, Inputs), Out), value_with(Args_new, Inputs,
   Args_old).
```

This rule handles the value of a function call. Given some argument expressions, they have a computed value, which is then used to call the function and is returned as the output.

```
value_with((A, B), (V1, V2), Args) :- eval_with((A, B), Args), value_with(A,
   V1, Args), value_with(B, V2, Args).
```

This rule deals with paired expressions. If you have two or more expressions to evaluate at the same time, (when dealing with multiple-argument functions, for example), it calculates the value of both and returns the paired result.

```
value_with(X, V, Args) :- eval_with(X, Args), where(X, Args, Expr),
   value_with(Expr, V, Args).
```

This rule handles the value of "where" expressions. If you have defined a where variable to define an expression, and that expression has a value, then the "where" variable also has that value.

```
value_with(X, V, Args) :- eval_with(X, Args), value(X, V).
value_with(N, N, Args) :- eval_with(N, Args), const(N).
value_with(N, N, Args) :- const_number(N), input(call(F, Args)).
```

These rules handle simple values. They mainly exist for correctness, so that a simple "value" can be used to calculate outputs.

```
value(mul(A, B), @to_num(V1) * @to_num(V2)) :- const_number(V1),
   const_number(V2), n_complex(mul(A, B)), value(A, V1), value(B, V2),
   const_number(@to_num(V1)*@to_num(V2)).
value(sub(A, B), @to_num(V1) - @to_num(V2)) :- const_number(V1),
   const_number(V2), n_complex(sub(A, B)), value(A, V1), value(B, V2),
   const_number(@to_num(V1)-@to_num(V2)).
value(add(A, B), @to_num(V1) + @to_num(V2)) :- const_number(V1),
   const_number(V2), n_complex(add(A, B)), value(A, V1), value(B, V2),
   const_number(@to_num(V1)+@to_num(V2)).
value((A, B), (V1, V2)) :- n_complex((A, B)), value(A, V1), value(B, V2).
value(N, N) :- n_complex(N), const_number(N).
```

These rules represent calculating values for non complex expressions. They work similarly to the `value_with` predicate described above.

```
eval_with(Expr, Inputs) :- input(call(F, Inputs)), rule(Index, F, Inputs,
   Expr), match(F, Index, Inputs).
```

This rule handles initialisation of the `eval_with` predicate. Semantically, if there is there an input to a matching rule, then you evaluate the body of that rule.

```
eval_with(A, Args) :- complex(A), eval_with(mul(A, B), Args).
eval_with(B, Args) :- complex(B), eval_with(mul(A, B), Args).
eval_with(A, Args) :- complex(A), eval_with(sub(A, B), Args).
eval_with(B, Args) :- complex(B), eval_with(sub(A, B), Args).
eval_with(A, Args) :- complex(A), eval_with(add(A, B), Args).
eval_with(B, Args) :- complex(B), eval_with(add(A, B), Args).
```

These rules define rule propagation of the `eval_with` predicate. If you are evaluating an arithmetic rule, then you have also have to evaluate all complex sub-expressions.

```
eval_with(Expr, Args) :- eval_with(X, Args), where(X, Args, Expr).
eval_with(Args_new, Args_old) :- complex(Args_new), eval_with(call(F,
   Args_new), Args_old).
eval_with(A, Args) :- complex(A), eval_with((A, B), Args).
eval_with(B, Args) :- complex(B), eval_with((A, B), Args).
```

Similarly, these rules define `eval_with` propagation for more complicated expressions. If you call a function, or have more than one expression together than you evaluate all complex sub-expressions

```
check_if_complex(Expr) :- eval_with(Expr, _).
```

This rule initialises `check_if_complex` predicates. If you have to evaluate an expression, you also have to check if it is complex

```
check_if_complex(Args) :- check_if_complex(call(F, Args)).
check_if_complex(A) :- check_if_complex((A, B)).
check_if_complex(B) :- check_if_complex((A, B)).
check_if_complex(A) :- check_if_complex(mul(A, B)).
check_if_complex(B) :- check_if_complex(mul(A, B)).
check_if_complex(A) :- check_if_complex(sub(A, B)).
check_if_complex(B) :- check_if_complex(sub(A, B)).
check_if_complex(A) :- check_if_complex(add(A, B)).
check_if_complex(B) :- check_if_complex(add(A, B)).
```

These rules define propagation of `check_if_complex` predicates, similarly to `eval_with` propagation.

```
complex((A, B)) :- complex(A), check_if_complex((A, B)).
complex((A, B)) :- complex(B), check_if_complex((A, B)).
complex(mul(A, B)) :- complex(A), check_if_complex(mul(A, B)).
complex(mul(A, B)) :- complex(B), check_if_complex(mul(A, B)).
complex(sub(A, B)) :- complex(A), check_if_complex(sub(A, B)).
complex(sub(A, B)) :- complex(B), check_if_complex(sub(A, B)).
complex(add(A, B)) :- complex(A), check_if_complex(add(A, B)).
complex(add(A, B)) :- complex(B), check_if_complex(add(A, B)).
```

These rules define propagation of `complex` predicates. The base case for a complex term is `complex(x0;x1;..;xN)` where N is the depth of the learned program. Then, if either sub-expression of an expression is complex, then the entire expression is complex.

```
n_complex(A) :- const(A), check_if_complex(A).
```

This rule introduces `n_complex` predicates. If a term is a constant, then it is not complex.

```
n_complex(call(F, Args)) :- n_complex(Args), check_if_complex(call(F, Args)).
n_complex((A, B)) :- n_complex(A), n_complex(B), check_if_complex((A, B)).
n_complex(mul(A, B)) :- n_complex(A), n_complex(B), check_if_complex(mul(A,
   B)).
n_complex(sub(A, B)) :- n_complex(A), n_complex(B), check_if_complex(sub(A,
   B)).
n_complex(add(A, B)) :- n_complex(A), n_complex(B), check_if_complex(add(A,
   B)).
```

These rules represent propagation of `n_complex` terms. If both sub-expressions of an expression are not complex, that that expression is also not complex.

3.2 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.
- I then add a constraint to remove any answer sets which do not produce the correct output against the given examples.

Not described here are my set of Skeleton Rules relating to valid arguments and valid expressions because they simply describe what is defined as valids (as per the Haskell syntax)

and I did not deem it important to include here.

Currently I can only learn very simple functions (without variables) because of errors in my Interpreter, relating to the ground rule predicates that are generated. However this should not be too difficult to fix.

Chapter 4

Evaluation and Future Work

4.1 Future Work

Before the end of this term (11th March) I hope to have a basic learning system in place. I will be able to learn programs with an arbitrary number of arguments, including lists, and these programs will be implemented using the languages features described in Chapter 3.
In addition, I will aim to have started implementation on the pre and post processing to convert input examples into ASP and the learned rules into Haskell.

During the Easter holidays I will first aim to spend some time trying examples and bug fixing on my existing code. Some thought will have to go into coming up with interesting and difficult examples.

After this I will start extending my tool to implement other Haskell language features. Most importantly is guards and let / where statements, allowing for more complicated conditionals and program structure. As part of this work I will have to introduce the generation of auxiliary functions to characterise this more complicated program structure.

After Easter I will have a lot of options for extending my tool:

- Implementing usage of Higher-order functions. This should theoretically not be too difficult but it is hard to say at this time.
- Various performance increases, perhaps reducing the grounding of the ASP encoding or extracting more work to the pre/post processing step. As part of this work, I could look into which optimisations applied to existing IFP tools are applicable to my tool (although at this time I think this will be quite limited due to the difference in approach).
- Development of a simple GUI or web interface to allow usage of my tool by interested third parties. While this would not have to be particularly complicated (see the MagicHaskeller web interface), it would still be work not directly improving the tool so I will have to evaluate the advantages of such an interface compared to just a command line tool closer to the time.
- Extend the input language from I/O examples to accept incomplete programs or specifications. This would be a useful feature to implement since it could be easier for the user to define a specification than a list of examples, however it could be challenging to implement due to the tight coupling of my encoding to the idea of examples as input.

4.2 Evaluation

To evaluate my tool I plan to use a combination of experimental evaluation and feedback from users.

For experimental evaluation I would first like to compare my tool to the two IFP systems described in section 2.2.2. This would involve running a number of example specifications through all three tools and time how long each one takes (or if they can compute an answer at all). At some point the existing systems will start to quickly degrade in performance and I hope that my tool will be able to last longer.
Another experimental evaluation that I think would be interesting is to attempt to learn the first year Haskell lab exercises and compare them to the model solutions. This could help highlight any syntax optimisations that could be implemented.

For user feedback it could be helpful to display my tool at the project demonstrations fair in May, so first and second year students can come and try my system, suggest improvements and try to find bugs which I may not have thought of. Before this I will have to have implemented at least some rudimentary interface.