Imperial College London
Department of Computing

Synthesis of Functional Programs using Answer Set Programming

James Thomas Rodden

Abstract

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo

velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

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.

2.3 The Target Language, Haskell

Chapter 3

The Initial Approach : 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 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.

3.1.1 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, mul(N, x1)) :- input(call(f, N)).

where(x1, N, call(f, x2)) :- input(call(f, N)).
where(x2, 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, mul(N, x1)) :- input(call(f, N)).

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

nesting each operation in its own clause.

3.2 Evaluating Rules

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

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. Since I am trying to learn Haskell programs, I need some way to match cases, either one of the base cases or the recursive case.
- `match_guard(F, Index, Inputs)` : This predicate is functionally similar to a guard expression is Haskell. Semantically, it represents the case in which the rule with the same Index matches, in comparison to the first `match` predicate being general over all rules.
- `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.

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

This rule represents generation of more input predicates. If you have to evaluate a function call, and the arguments have value `Inputs`, then you take the function call argument `Inputs` as additional function input.

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

These rules handle choosing which rule to use. A rule matches some inputs if there exists a rule with those inputs which does not match a a rule before it.

```
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.

3.3 A Worked Example : Greatest Common Divisor

To illustrate the interpreter, I will walk through a simple example - Euler's algorithm to calculate the Greatest Common Divisor. The Haskell definition of this function is :

```
gcd x y
       | x == y = x
       | x > y = gcd(x - y, y)
       | x < y = gcd(x, y - x)
```

You may notice this is not the typical definition of Euler's algorithm, as it does not make use of modulo. This is because my current language bias does not support the modulo operator.

I represent these Haskell rules in ASP as :

```
%rule(1, gcd, (A, B), A) :- input(call(gcd, (A, B))).

%rule(2, gcd, (A, B), call(gcd, (x1, B))) :- input(call(gcd, (A, B))).
%where(x1, (A, B), sub(A, B)) :- input(call(gcd, (A, B))).

%rule(3, gcd, (A, B), call(gcd, (A, x2))) :- input(call(gcd, (A, B))).
%where(x2, (A, B), sub(B, A)) :- input(call(gcd, (A, B))).
```

And the corresponding `match` rules are :

```
%match_guard(gcd, 1, (A, B)) :- A == B, input(call(gcd, (A, B))).
%match_guard(gcd, 2, (A, B)) :- A > B, input(call(gcd, (A, B))).
%match_guard(gcd, 3, (A, B)) :- A < B, input(call(gcd, (A, B))).
```

3.3.1 A Simple Input

To start with, I will work through a simple example which matches the base case, `gcd(3, 3)`. This input is represented in ASP as the term `input(call(gcd, (3, 3)))`, meaning function `gcd` is called with `(3, 3)` as the arguments.

The first predicates generated are related to matching. As `(3 == 3)`, the first `match_guard` rule is applied and a the term `match_guard(gcd, 1, (3, 3))` is added to the output.

The rule next applied is

```
match(F, Index, Inputs) :- not smaller_match(F, Index, Inputs), rule(Index,
   F, Inputs, _), match_guard(F, Index, Inputs).
```

As the relevant ground `rule` and `match_guard` terms exist, and there are no smaller matches, (as 1 is the smallest Index), I generate a `match(gcd, 1, (3, 3))` term for the output.

Next, the tool has to decide what to evaluate, by generating `eval_with` terms. The rule applied is :

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

Because the tool knows the ground `rule` term `rule(1, gcd, (3, 3), 3)`, it `match` that rule with the term `match(gcd, 1, (3, 3))` and it knows the `input` term `input(call(gcd, (3, 3)))`, it can apply this rule and get the expected term, `eval_with(3, (3, 3))`, as part of the output. Semantically, this makes sense, as we want to evaluate the body of the called function, with respective arguments.

Now, the tool can work out values, through application of the rule

```
value_with(N, N, Args) :- eval_with(N, Args), const(N).
```

The tool just generated the term `eval_with(3, (3, 3))`, and it knows that 3 is a constant, so it can output that `value_with(3, 3, (3, 3))`.

Finally, we can generate output. Using the rule

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

In the same way to earlier rule applications, the tool knows the relevant ground `rule(1, gcd, (3, 3), 3)` and `match(gcd, 1, (3, 3))` terms, and we just generated the `value_with(3, 3, (3, 3))` term. Therefore, the tool can return `output(call(gcd, (3, 3)), 3)`, as expected.

3.3.2 A More Complex Example

Now I will detail a more complicated example, where the tool visits where rules and more complicated expressions. As input, I will use `gcd(9, 6)`, which is represented in ASP as the term `input(call(gcd, (9, 6)))`.

Once again, initially the tool generates `match` terms using the rule :

```
match(F, Index, Inputs) :- not smaller_match(F, Index, Inputs), rule(Index,
   F, Inputs, _), match_guard(F, Index, Inputs).
```

This time, it generates the ground term `match(gcd, 2, (9, 6))`, as $(9 > 6)$ it hits the second rule guard. It then attempts to determine which rules to evaluate, using the rule :

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

This rule generates the term `eval_with(call(gcd, (x1, 6)), (9, 6))`. This is where is example diverges from the first. We first have to evaluate the arguments, because they are complex.

I know these rules are complex because we have checked them. Using this rule :

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

This rule generates the term `check_complex(call(gcd, (x1, 6)))`. Then using the next rule I check the complexity of the arguments

```
check_if_complex(Args) :- check_if_complex(call(F, Args)).
```

This then generates the term `check_complex((x1, 6))`. It then checks each argument individually, using the rules

```
check_if_complex(A) :- check_if_complex((A, B)).
check_if_complex(B) :- check_if_complex((A, B)).
```

Now, the tool knows that `x1` is defined as complex, so we can start rebuilding the argument expression as complex. Using the rule

```
complex((A, B)) :- complex(A), check_if_complex((A, B)).
```

This tells the tool to include the term `complex((x1, 6))`. Now, having checked that the arguments are complex, we can evaluate them, using

```
eval_with(Args_new, Args_old) :- complex(Args_new), eval_with(call(F,
   Args_new), Args_old).
```

The tool now knows to `eval_with((x1, 6), (9,6))`. To do this, it evaluates all of the complex sub-expressions, using the rule

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

This produces the term `eval_with(x1, (9, 6))`. Because it has to now evaluate a where variable, it chooses the rule

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

As I have already defined the body of the where rule referenced by `x1` to be `where(x1, (A, B), sub(A, B))`, the tool then takes the ground version of this definition and applies the rule to get the term `eval_with(sub(9, 6), (9,6))`.

This rule is why the tool needs to pass the relevant arguments around as it evaluates terms. If the `(9, 6)` was missing, then the tool would not know which ground version of the rule to use. The tool has now reached the end of evaluating, because `eval_with(sub(9, 6), (9, 6))` is not complex. In the same way the tool checked for complexity, checking if an expression is not complex relies on `check_if_complex` terms. Once again, I use the rule

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

Which introduces the term `check_if_complex(sub(9, 6))`. As before, this rule then propagates, generating `check_if_complex(9)` and `check_if_complex(6)` terms. Now, as 9 and 6 are constants, they are not complex, as defined by this rule

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

Now the tool knows `n_complex(9)` and `n_complex(6)`. It then uses the rule

```
n_complex(sub(A, B)) :- n_complex(A), n_complex(B), check_if_complex(sub(A,
   B)).
```

Which produces the term `n_complex(sub(9, 6))`, as expected. This term can now have its value calculated, as it is not complex. Using the rule

```
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)).
```

This generates the term `value(sub(9, 6), 3)`, as it already knows `value(9, 9)` and `value(6, 6)`, as they are constants. Now, the tool can work out the value of the where variable, `x1`, using this rule :

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

This gives the tool the term `value_with(x1, 3, (9, 6))`. Now the tool has almost calculated the value of the function call arguments, all it has to do it use this rule

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

Which combines the terms `value_with(x1, 3, (9, 6))` and `value_with(6, 6, (9, 6))`, to get `value_with((x1, 6), (3, 6), (9, 6))`.

Remember that the overall goal of this was to be able to evaluate the term `eval_with(call(gcd, (x1, 6)), (9, 6))`. Now, this is possible as the tool has generated the value of the arguments. The tool applies the following rule

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

Which generates a new `input(call(gcd, (3, 6)))` term, effectively starting the process again for a new input. This input is processed in a very similar way to the last one, with the only difference being that the third rule is matched, as `3 < 6`. This means that the rule body to be evaluated produces the term `eval_with(call(gcd, (3, x2)))`.

Once again, this evaluation proceeds by attempting to calculate the value of the arguments, more specifically the complex one, `x2`. Using the definition, `where(x2, (A, B), sub(B, A))`, the tool can generate the ground term `value_with(x2, 3, (3, 6))`. This term is then combined with the non complex argument to produce the term `value_with((3, x2), (3, 3), (3, 6))`. Then, it generates a new input predicate again by applying the rule

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

Now, we reach the base case, as the input generated is `input(call(gcd, (3, 3)))`. As I explained in the first example, this input generates an output term in the Answer Set of `output(call(gcd, (3, 3)), 3)`. It reaches this conclusion by matching to the base case, which is defined to return the first argument.

With this `output` term, the tool can now traverse back up the call stack, to compute a final result. By applying the following rule

```
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).
```

I can generate the term `value_with(call(gcd, (3, x2)), 3, (3, 6))`. This then allows me to use the rule :

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

Which gives a corresponding `output` term `output(call(gcd, (3, 6)), 3)`. Once again, I can use this output to generate the term `value_with(call(gcd, (x1, 6)), 3)`, and then use this term to generate the final `output` term, `output(call(gcd, (9, 6)), 3)`, as expected!
TODO: PUT DIAGRAM HERE.

Chapter 4

Learning from Examples

This chapter will detail how I use the interpreter discussed in chapter 3 to perform learning. The tool enumerates all possible rules, then chooses the ones which cover all of the examples.

4.1 Additional Rules

Together with my interpreter, I need the following addition predicates and rules :

- `example(Input, Output)` : This predicate represents an Input / Output pair.
- `choose(R, N)` and `choose_where(N)` : These predicates represent the choice of a rule, with depth R, that covers all of the examples.
- `input(In):- example(In, _).` : This rule generates the initial inputs for my interpreter.
- `:- not output(In, Out), example(In, Out).` : This constraint represents that you cannot have an example which does not produce a matching output. In other words, this rule rule will removes rules which do not cover the examples.

4.2 Skeleton Rules

To know what rules are possible to learn, I enumerate all possible combinations of rules, to provide a set for the learning task to choose from. While it may seem that the possible search space is very large, this is only partly true, due to the optimisations allowed by use of `where` clauses.

Each skeleton rule has one of the following formats:

```
rule(R, F, Args, Expr):- input(call(f, Args)), choose(R, N).

where(Var, Args, Expr):- input(call(f, Args)), choose_where(N).
```

Where `Expr` is one of the possible rule bodies.

4.2.1 Choice Rules

To run the interpreter on all possible rule combinations, I make use of ASP choice rules. For example, the statement :

```
1 {
choose(R, 2..5;11..16),
choose(R, 1;6..10, C0) : expr_const(C0)
} 1 :- num_rules(R).
```

Represents the learning task choosing exactly one of the skeleton rules for each of the possible program rules (or recursive case). Additionally, I need a choice rule for every potential where rule :

```
0 {
choose_where(17..19;24..28;30),
choose_where(20..23;29, C1) : expr_const(C1)
} 1.
```

Here, it is not guaranteed for a where rule to be chosen, as they may not all be necessary in different learning tasks.

4.2.2 Rule combinations

The depth of the search space is reliant on three main factors : number of arguments, range of allowable constants and target language complexity. My initially small target language means that I only have to enumerate over addition, subtraction, multiplication and function calls, but as I add more expressions (i.e boolean functions), the size of the skeleton rules increases respectively.

Similarly, the number of arguments of the target function increases as I have to enumerate all possible pairs. For example, for a simple predicate like addition I need to include : (where X, Y are arguments, C is an arbitrary learned constant, and x1 and x2 are where variables)

- `add(X, X)`
- `add(X, Y)`
- `add(X, C)`
- `add(X, x1)`
- `add(X, x2)`
- `add(Y, Y)`
- `add(Y, C)`
- `add(Y, x1)`
- `add(Y, x2)`
- `add(C, x1)`
- `add(C, x2)`
- `add(x1, x2)`

Even in this simple example, there are 12 possible rules. As addition is commutative, I have omitted any rules where the ordering is reversed. More optimisations like this can be seen in the next section. A full list of the skeleton rules generated for both one and two argument functions can be seen in the appendix.

4.2.3 Learning Match Rules

Until now, I have been assuming that the tool knows about the specific rule guard matching behaviour before learning takes place. While this is helpful initially, for this tool to work I need to also learn the respective match rules. Luckily, this is not particularly complicated.

As guard rules have to return booleans, this reduces the number of operations to the integer comparators, `equals (==)` and `less than (<)`. I choose to omit `greater than (>)` as I can replace all occurrences of it with `(<)` without loss of generality.

Using these operations, the skeleton rules for match terms are given by :

```
match_guard(gcd, R, (N0, N1)) :- N0 == C1, input(call(gcd, (N0, N1))),
   choose_match(R, 1, C1).
match_guard(gcd, R, (N0, N1)) :- N1 == C1, input(call(gcd, (N0, N1))),
   choose_match(R, 2, C1).
match_guard(gcd, R, (N0, N1)) :- N0 == N1, input(call(gcd, (N0, N1))),
   choose_match(R, 3).
match_guard(gcd, R, (N0, N1)) :- N0 < N1, input(call(gcd, (N0, N1))),
   choose_match(R, 4).
match_guard(gcd, R, (N0, N1)) :- N1 == N0, input(call(gcd, (N0, N1))),
   choose_match(R, 5).
match_guard(gcd, R, (N0, N1)) :- N1 < N0, input(call(gcd, (N0, N1))),
   choose_match(R, 6).
```

To choose these rules, I once again make use of a choice rule, which makes sure we choose exactly as many skeleton match rules as the user defines.

```
1 {
choose_match(R, 3;4;5;6) ,
choose_match(R, 1;2, C0) : expr_const(C0)
} 1 :- num_match(R).
```

4.3 Multiple Solutions and Optimisation

While learning, it is not uncommon to have multiple solutions, usually due to multiple semantically equivalent base cases. I may get two answer sets as output, one having learned `rule(1, F, 0, 1)`, and the other with `rule(1, F, 0, 0+1)`.

To attempt at dealing with this, I have implemented a basic optimisation system, by prioritising rules with shorter bodies. In general, rules with lower rule number are shorter, so I used the minimisation rule :

```
#minimise [choose(_, N)=N, choose(_, N, _)=N ].
```

Which prioritises rules with lower rule numbers.

As a second optimisation, I wanted to prefer less complicated results, and answer sets with fewer `where` clauses represent less complicated functions, as the depth of the function is lower. Because of this, I use the minimisation rule :

```
#minimise[choose_where(R)=1, choose_where(R, C)=1].
```

4.4 A Worked Example : Learning GCD

In the last chapter, I went through the steps that the interpreter takes to produce results when given Euler's algorithm to calculate the greatest common divisor. In this section, I will detail how I then learn that program. As a reminder, this function is defined in Haskell as :

```
gcd x y
       | x == y = x
       | x > y = gcd (x - y) y
       | x < y = gcd x (y - x)
```

As input to the learning task I need to specify some examples. I initially chose the examples

```
example(call(gcd, (1, 1)), 1).
example(call(gcd, (2, 1)), 1).
example(call(gcd, (4, 3)), 1).
example(call(gcd, (3, 6)), 3).
example(call(gcd, (9, 6)), 3).
```

Because they seem to cover all of the cases I need to learn. In addition to this, I will need to enumerate all of the possible `rule` and `where` bodies as part of the skeleton rules. I will not list the entire set of skeleton rules here, but instead highlight some which will be in use later :

```
rule(R, gcd, (N0, N1), N0) :- input(call(gcd, (N0, N1))), choose(R, 1).
rule(R, gcd, (N0, N1), N1) :- input(call(gcd, (N0, N1))), choose(R, 2).
rule(R, gcd, (N0, N1), sub(N0, N1)) :- input(call(gcd, (N0, N1))), choose(R,
   23).
rule(R, gcd, (N0, N1), call(gcd, (N1, x0))) :- input(call(gcd, (N0, N1))),
   choose(R, 30).
rule(R, gcd, (N0, N1), call(gcd, (x1, N0))) :- input(call(gcd, (N0, N1))),
   choose(R, 81).

where(x0, (N0, N1), sub(N0, N1)) :- input(call(gcd, (N0, N1))),
   choose_where(53).
where(x1, (N0, N1), sub(N1, N0)) :- input(call(gcd, (N0, N1))),
   choose_where(77).

match_guard(gcd, R, (N0, N1)) :- N0 == N1, input(call(gcd, (N0, N1))),
   choose_match(R, 3).
```

```
match_guard(gcd, R, (N0, N1)) :- N0 < N1, input(call(gcd, (N0, N1))),
   choose_match(R, 4).
match_guard(gcd, R, (N0, N1)) :- N1 < N0, input(call(gcd, (N0, N1))),
   choose_match(R, 6).
```

After running the learning task with these inputs, I get as output the terms :

```
choose(1, 1).
choose(2, 23).
choose(3, 2).
```

```
choose_match(1, 4).
choose_match(2, 6).
choose_match(3, 3).
```

These rules represent the learned Haskell program:

```
gcd x y
       | x == y = y
       | x > y = x - y
       | x < y = x
```

What is interesting is that this is not a correct implementation of Euler's algorithm! However, this result is still returned as it covers all of the examples. For example, the example `example(call(gcd, (9, 6)), 3)` generates a `input(call(gcd, (9, 6)))` term. This term is then ran through the interpreter with the chose rules, and returns `output(call(gcd, (9, 6)), 3)`. As this does not contradict the example, those rules are valid.

As an attempt to fix this problem, I can add some extra examples. By adding these two examples :

```
example(call(gcd, (4, 7)), 1).
example(call(gcd, (9, 3)), 3).
```

These examples are cases where the previous result would not work, and help generalise the target function.

After running these examples as input, I get the following terms as a result :

```
choose(1, 1).
choose(2, 81).
choose(3, 30).
choose_where(77).
choose_where(53).
choose_match(1, 3).
choose_match(2, 4).
choose_match(3, 6).
```

These terms translate to the haskell program :

```
gcd x y
        | x == y = x
        | x > y = gcd y x0
        | x < y = gcd x1 x

        where x0 = x - y
        where x1 = y - x
```

As required.

4.5 Performance Issues

Unfortunately, this learning task suffers from extreme difficulty scaling. While the number of skeleton rules is relatively small, the problem comes when considering combinations of `rule` and `where` terms.

For the example described above, the output ground program is 366879 lines long, and takes over 500 seconds to run, which is far too long, especially for someone using a front-end user interface.

4.5.1 Potential Optimisations

However, there are many potential optimisations I could make here, to increase performance. The first involves limiting the range of available constants. As default, I restricted integer constants to 0 - 10, defining this using the term `const_number(0..10)`. However, it may be more efficient to limit the maximum integer size to the largest constant found in the examples. This would allow for great increase in performance, especially if working on expressions such as lists, where the value inside of the list usually does not matter.

However, this would limit internal values inside the program. If a program requires an internal variable whose value is greater than any of the examples, then my tool would fail. But, it is important to consider if there are very many programs in which this would occur, especially considering my limited language bias.

Other potential optimisations include

- Removing redundant skeleton rules
- Using clingo's built in arithmetic for simple expressions.
- Finding the optimal number of potential where rules.

In the end, I did not choose to implement any of these optimisations, and instead opted to implement an entirely new approach, as detailed in the next chapter.

Chapter 5

A Second Approach : Constraint Based Learning

As seen in the previous chapter, my initial approach suffered from significant difficulty scaling. Even simple two argument functions took upwards of 130 seconds to complete. This was mainly due to the sheer size of the ground learning task, and the complexity from combinations of rule and where predicates.

5.1 Top Down Vs. Bottom Up

The main issue with the interpreted approach is that it is bottom-up. To learn the output of a rule, we must first iterate down the expression tree, calculate the value of each simple sub-expression, then iterate back up the tree combining values until we know a value for the entire rule body. My second approach overcomes this issue by implementing a top down approach.

The idea behind this new approach is simple. By maintaining an "equality constraint", I keep track of what each expression is supposed to be equal to (as defined by the input examples). Then, as the program iterates down the expression tree, it fails if it ever finds some easily provable equality failure, i.e $1 == 2$.

For example, if I know that `call(f, 2)== 5` and that the body of function f is `2*X + 1`, then I can deduce that

$$\begin{aligned}(2 * 2) + 1 &== 5\\ (2 * 2) &== 4\\ 2 &== 2\end{aligned}$$

So there are no contradictions.

However, if instead we have an example stating that `call(f, 2)== 6`, with the same function, then instead we get

$$\begin{aligned}(2 * 2) + 1 &== 6\\ (2 * 2) &== 5\end{aligned}$$

Which fails because 5 is not a multiple of 2.

5.1.1 Dealing with termination

One issue with this new approach is that it does not automatically handle programs which do not terminate. Whilst these programs do not stop clingo from finding a solution, they are incorrect as the constraint is never met.

To deal with this, I have to check input examples for termination. While this is typically an undecidable problem, for my small target language it is decidable and computable. To represent this in ASP, I first needed a way to represent the next step of execution of an expression. For example, I can say that `next_step(add(1, mul(2, 3)), mul(2, 3))`, meaning that I next evaluate mul(2, 3).

I then define termination as :

- If an expression is simple, containing only constants, then it terminates.
- If the next step of an expression terminates, then that expression also terminates.

This approach is fairly efficient as it as another top down approach, and generates a similar number of rules in the ground program as the constraint checking rules.

5.2 ASP Representation

I represent this approach using the following ASP predicates.

- `eq(Expr, Val)` represents an equality. Expr, when evaluated, should equal Val.
- `is_call(call(F, Expr)` represents if a function is called. Used to generate more ground skeleton rules.
- `terminates(Expr)` represents if an expression terminates.
- `next_step(A, B)` represents that B is the expression executed after executing A.

Now, I use these predicates in the following rules.

```
eq(In, Out) :- example(In, Out).
```

This rule generates initial equality constraints given by the examples.

```
:- eq(N1, N2), const_range(N1), const_range(N2), N1 != N2.
```

This rule constrains equality on constants. The tool should fail if two different constants are equal.

```
:- eq(mul(0, B), Val), Val > 0.
:- eq(mul(A, 0), Val), Val > 0.
:- eq(mul(A, B), Val), const_number(B), B > 0, Val #mod @to_num(B) != 0.
:- eq(mul(A, B), Val), const_number(A), A > 0, Val #mod @to_num(A) != 0.
```

These constraints handle edge cases when dealing with multiplication. If multiplying any expression by 0, then it should be equal to 0, and if multiplying two things together, then the answer should be a multiple of them both.

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

This constraint handles termination. The tool fails if an input example does not terminate.

```
is_call(call(F, Expr)):- eq(call(F, Expr), Out).
```

This rule generates `is_call` predicates, which are used to generate more ground instances of skeleton rules.

```
eq(Expr, Val) :- rule(Index, F, Arg_Expr, Expr), match(F, Index, Arg_Expr),
   eq(call(F, Arg_Expr), Val).
```

This rule handles propagation of equality constraints through function calls. If a called function is equal to some value, then the body of the function (with correct arguments) is also equal to that value.

```
eq(A, Val - B) :- eq(add(A, B), Val), const_number(B), B <= Val.
eq(B, Val - A) :- eq(add(A, B), Val), const_number(A), A <= Val.
eq(A, Val / B) :- eq(mul(A, B), Val), const_number(B), B > 0.
eq(B, Val / A) :- eq(mul(A, B), Val), const_number(A), A > 0.
eq(A, Val + B) :- eq(sub(A, B), Val), const_number(B), const_number(Val).
eq(B, Val + A) :- eq(sub(A, B), Val), const_number(A), const_number(Val).
```

These rule specify generation of equality predicates with arithmetic, through use of the opposite operations. Addition terms in the head of the rule are necessary to handle edge cases such as division by zero.

```
terminates(Expr) :- next_step(Expr, Val), const(Val).
terminates(Expr) :- next_step(Expr, Term_Expr), terminates(Term_Expr).
```

These rules define termination, as described above. If an expression is a constant, then it terminates, or if the next step of an expression terminates then that expression also terminates.

```
next_step(e, In):- example(In, Out).
```

This rule generates the initial `next_step` predicates. If there exists an example input, then it is the next step of some arbitrary term `e`.

```
is_next_step(X) :- next_step(_, X).
```

This rule exists to reduce the grounding. If any expression is a next step, we generate an `is_next_step` predicate, used in the remaining `next_step` rules.

```
next_step(call(F, Args), Expr) :- is_next_step(call(F, Args)), rule(Index, F,
    Args, Expr), match(F, Index, Args).
```

This rule handles the next step of function calls. The next step of a function call is the body of that function.

```
next_step(add(A, B), B) :- is_next_step(add(A, B)), const_number(A).
next_step(add(A, B), A) :- is_next_step(add(A, B)), const_number(B).
next_step(mul(A, B), B) :- is_next_step(mul(A, B)), const_number(A).
next_step(mul(A, B), A) :- is_next_step(mul(A, B)), const_number(B).
next_step(sub(A, B), A) :- is_next_step(sub(A, B)), const_number(B).
next_step(sub(A, B), B) :- is_next_step(sub(A, B)), const_number(A).
```

These rules generate `next_step` predicates for arithmetic. The next step of an arithmetic expression is the argument that is not constant.

5.3 Learning

The actual learning task operates in a similar way to my initial approach. By enumerating all possible rule bodies, I can use a choice rule to try generating answer sets with each one, keeping the answer sets which are satisfiable. However, because I am no longer using `where` clauses, the skeleton rules now contain full bodies. This means that unfortunately the number of skeleton rules becomes large, numbering in the thousands for even simple tasks. To avoid this scaling poorly once again, I decided to implement a number of simple optimisations.

5.3.1 Using inbuilt arithmetic

As part of my first approach, I decided to represent arithmetic with my own predicates `add(A, B)`, `sub(A, B)` and `mul(A, B)` because I needed to evaluate expressions that the grounder would not be able to compute (i.e the value of function calls or where variables).

However, in my new approach I decided to make partial use of the clingo inbuilt arithmetic. If inside function arguments, or the rule body has no function calls at all, then I know all sub expressions will be arithmetic and I make use of the simple `+, -` and `*`.

The main advantage of this approach is that it vastly reduces the ground output. As the inbuilt operations are computed by the grounder, if two different expressions compute the same output number, then they are not repeated in the ground output. For example, `X + X` and `2 * X` are semantically equivalent, so only produce one output rule.

5.4 A Worked Example : Greatest common divisor

Once again I will use Euler's algorithm for the Greatest Common Divisor to illustrate this new approach. Because my tool still does not have modulo as part of its target language, I will still attempt to learn the simplified definition :

```
gcd x y
       | x == y = x
       | x > y = gcd(x - y, y)
       | x < y = gcd(x, y - x)
```

As input, I will be using the following examples :

```
%rule(1, gcd, (A, B), A) :- input(call(gcd, (A, B))).

%rule(2, gcd, (A, B), call(gcd, (x1, B))) :- input(call(gcd, (A, B))).
%where(x1, (A, B), sub(A, B)) :- input(call(gcd, (A, B))).

%rule(3, gcd, (A, B), call(gcd, (A, x2))) :- input(call(gcd, (A, B))).
```

Which cover both cases $(X < Y)$ and $(X > Y)$ while also not being too simplistic.

After running the learning task, the resulting Answer Set contains the terms:

```
choose(1,2).
choose(2,150).
choose(3,190).
```

These terms correspond to the following skeleton rules :

```
rule(R, gcd, (N0, N1), N0) :- is_call(call(gcd, (N0, N1))), choose(R, 2).
rule(R, gcd, (N0, N1), call(gcd, ((N0 - N1), N1))) :- is_call(call(gcd, (N0,
   N1))), choose(R, 150).
rule(R, gcd, (N0, N1), call(gcd, (N1, N0))) :- is_call(call(gcd, (N0, N1))),
   choose(R, 190).
```

Which then corresponds to the Haskell program :

```
gcd x y
       | x == y = x
       | x > y = gcd (x - y) y
       | x < y = gcd y x
```

What is interesting is that whilst it has not learned the exact target program, the learned program is still correct. This is due to the optimisations I have implemented preferring rules with shorter bodies.

To see why I get this result, it is useful to look at the corresponding `eq` predicates for each example.

The example `example(call(gcd,(8,12)),4)` produces the terms :

```
eq(call(gcd,(8,12)),4).
eq(call(gcd,(12,8)),4).
eq(call(gcd,(4,8)),4).
eq(call(gcd,(8,4)),4).
eq(call(gcd,(4,4)),4).
eq(4, 4).
```

```
next_step(e,call(gcd,(8,12))).
next_step(call(gcd,(8,12)),call(gcd,(12,8))).
next_step(call(gcd,(12,8)),call(gcd,(4,8))).
next_step(call(gcd,(4,8)),call(gcd,(8,4))).
next_step(call(gcd,(8,4)),call(gcd,(4,4))).
next_step(call(gcd,(4,4)),4).
```

Because these terms are all created without constraints failing, the respective rule bodies are returned as a solution.

5.5 Performance

While this new approach does perform better than the old one, it still suffers from a lot of the same issues limiting performance.

Because of the large number of combinations of skeleton rules, they can still grow very large very quickly, lowering the performance of my tool. In addition, expanding the language bias to different types also has an adverse effect on this.

5.5.1 Reducing the language bias

As a way to deal with the explosive expansion of skeleton rules, I decided to implement a way to help contain this by artificially limiting the language bias.

By removing operations from the bias that I know will not be used in the output functions, I can remove a large number of skeleton rules that will have no effect on the output of the learning task. In a similar way, I can limit learning to only tail recursive programs, meaning that my skeleton rules only use inbuilt arithmetic operations, further increasing performance.

Of course, this method for improving performance is limited by the knowledge of the user. If they have no idea what the output function will look like, this is completely unhelpful.

Chapter 6

Front end implementation : Building a working UI

6.1 User's Manual

6.2 Used Technologies

6.2.1 Play Framework

6.3 User Feedback and Evaluation

Chapter 7

Critical Evaluation

7.1 Testing

7.1.1 One Argument Programs

Factorial

7.1.2 Two Argument Programs

Tail Recursive Factorial

7.2 Comparison to Existing Tools

Chapter 8

Conclusions and Future Work

8.1 Conclusions

8.1.1 What worked

8.1.2 Areas for improvement

8.2 Future Work

8.2.1 Learning Improvements

8.2.2 UI Features