Imperial College London
Department of Computing

Synthesis of Functional Programs using Answer Set Programming

James Thomas Rodden

Abstract

Acknowledgements

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.

Answer Set Programming 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,

1.1 Motivation

My project introduces a new approach to IFP, through the use of ASP. Being oriented towards difficult search problems, ASP has been successfully applied to similar problems in the fields of planning, robotics and ontologies. Since one approach to IFP categorises the learning problem as a search problem over the range of possible programs, it seemed natural to apply ASP in this area.

This project is does not just develop a new approach to IFP, but makes one that is easier to use as well. As they are developed by academics, existing IFP are either lacking user interfaces or have rudimentary ones, often making usage or translating results difficult. Through exploring ways to make the UI easier to use, it becomes possible to start considering possible application of this technology.

I focused my project as a learning tool. It is not uncommon for a new Haskell student to be unfamiliar with recursion, having no experience in writing programs in a functional style. An easy to use IFP system could help beginners by allowing them to experiment with creating different programs, helping them become more familiar with the Haskell syntax.

1.2 Contributions

These motivations are achieved through the implementation of my tool. I have created a learning system consisting of a responsive, user-friendly web based user interface, a core learning algorithm based on ASP and a back end system used to communicate between the two. This system takes a set of input / output examples and uses these examples to subsequently learn

and return a valid Haskell function.

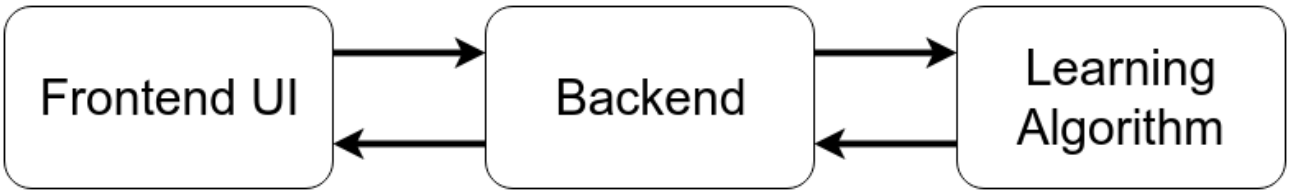

Figure 1.1: Basic tool structure

My ASP learning algorithm implements two different approaches. The first, detailed in Chapters 3 and 4, makes use of a Haskell interpreter implemented in ASP which runs Haskell programs re-written as ASP rules. This interpreter is then used in the learning task to determine which possible functions return the correct result when ran on the example inputs. The most optimal of these functions is returned as the learned result.

The second approach, detailed in Chapter 5, maintains an "equality constraint" for each input example pair. This constraint is initialised with the fact that the result of a function call with a specific input should be equal to the respective output. Then, the constraint is maintained as the function call is evaluated, and fails if an obvious contradiction is reached (i.e. $1 = 2$). Functions which do not fail this constraint over all input examples are returned as possible solutions.

A second approach is necessary due to performance issues with the first approach. Using an interpreter means evaluating the value of all sub-expressions of a function, gradually combining expressions until an output is produced. This is inefficient as atoms have to be generated both when working out which expressions to evaluate, and when evaluating them. In comparison, the constraint-based approach fails as soon as a contradiction is reached, eliminating the need to produce an overall output to a function.

The user interface is built to allow for easy experimentation and testing. Users can enter examples with any number of arguments, getting quick results in the form of the learned program. Then, if this program is incorrect the user can add a new counter-example or modify any examples that may be incorrect. In addition, if the user is unsure about a generated program then they can add examples with no respective output, which are then auto-completed. If the result of this auto completion is as expected, then the generated function is likely correct.

These two parts are linked by a backend, written in Java. This backend manages all parts of the learning that would be too difficult to implement in either the frontend or the ASP, including :

- Generating skeleton rules enumerating all possible programs.
- Converting inputs into valid ASP.
- Running the learning task in parallel.
- Converting the output from ASP rules to valid Haskell.
- Running the generated Haskell to auto-complete empty examples.

Chapter 2

Background

2.1 Common Terminology

Before detailing the background relating to ASP, a list of common terms that will be used throughout can be found in table 2.1.

2.2 Answer Set Programming

2.2.1 Negation As Failure

Negation as failure is an inference rule used in non-monotonic logic programs to derive atoms of the form `not p`. This derivation is defined as

> "`not p` can be inferred if every possible proof of `p` fails." [?]

In more simple terms, this means that you can assume that `not p` holds if you cannot prove `p`. As an example, consider the program

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

As `q` cannot be proven, we can instead prove `not q`, which is then used to prove `p`.

2.2.2 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, and 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}`.

Table 2.1: Common Logic Terms

Constant	A sequence of characters starting with a lower case character or number. Typically denoted `a, b, c, ... p, q, r, ....`
Variable	A character or sequence of characters starting with an upper case letter, representing some range of possible constants. Typically denoted `A, B, C, ... X, Y, Z, ....`
Term	Either a constant or a variable
Predicate	A character or sequence of characters representing some relation between Constants or Variables. For example, the predicate `on, above, below`.
Atom	A predicate applied to some terms. `on(X, Y), above(a, b), p(X), q(Z)` are all examples of atoms.
Literal	Either an Atom or an Atom preceded by the `not` operator, representing Negation By Failure.
Rule	A line of the form `head :- body` where `head` is an atom and `body` is a list of literals.
Program	A collection of rules

Grounding

Before you can solve an extended logic program using Answer Sets you must first ground it. A rule is said to be "ground" if all of its variables are replaced by every combination of possible constants. Then, to calculate the grounding of a program P you replace each rule in P by every ground instance of that rule. For example, the 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).
```

This is calculated as the grounding for this program, because the variables `(X, Y)` in the rule `q(X):- p(X, Y), not q(Y).` are replaced by all possible combinations of the constants `(1, 2)`. These combinations are, specifically :

```
{ X = 1, Y = 1 }
{ X = 1, Y = 2 }
{ X = 2, Y = 1 }
{ X = 2, Y = 2 }
```

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).` Unsafe because the variable `X` does not appear in the body.
- `p(Z):- not q(X, Y), r(X, Y).` Unsafe because the variable `Z` does not appear the body of the rule.
- `p(X):- q(X), not r(Y).` Unsafe because the variable `Y` does not appear positively in the body.

Least Herbrand Models

For a normal logic program P, the following features are defined as follows:

Herbrand Base	The set of all ground atoms of a program.
Herbrand Interpretation	An assignment on the Herbrand Base, specifying which ground atoms are true and which are false. A rule is satisfied by an interpretation if, when all atoms of the body are specified as true in the interpretation, then so is the head of the rule.
Herbrand Model	A Herbrand Interpretation where every rule in the program is satisfied by that interpretation. 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, consider the program P:

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

P has the Herbrand Base `{p, q}`.

Because listing if each atom is true or false is overly verbose, instead we write the Herbrand Interpretation as a set of atoms, including the ones specified as true. One Herbrand Interpretation of P is `{p, q}`, but this interpretation is not a model because neither rules are satisfied by it.

The Herbrand Interpretations `{p}` and `{q}` are both Herbrand Models. `{p}` is a model because it satisfies the both rules - the first as `not q` holds by NBF and `p` is in the model, and the second as `(not p)` does not hold so `q` is not needed. `{q}` is a Herbrand Model for the same reasons, reversed. The first line is satisfied because the body `not q` does not hold, and the second line is satisfied as `not p` holds, and `q` is in the interpretation.

2.2.3 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).
```

First, try X = `{q(1)}`. The reduct is calculated as :

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

The Least Herbrand Model of this reduct is `{p(1, 2), q(1)}`. Therefore X is not an Answer Set of P.

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}`. This is because the Answer Set `{p}` fails the constraint, as it contains `p` and does not contain `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 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 { on(A, B), on(B, A)} 1.` represents `A` can be on `B`, or `B` can be on `A`, 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}`, with optimisation 1.

2.2.4 ASPAL

ASPAL is an ASP algorithm which computes the solution of an 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.

These skeleton rules are defined by "mode declarations", terms which specify which rules are allowed to appear in the heads of rules, and which are allowed to appear in the bodies. The predicate `modeh` specifies the head declaration, while `modeb` specifies the body.

We next rule out Answer Sets which do not fit the examples by adding a goal rule and a constraint, of the format (where $e_1^+, \ldots, e_n^+$ are the positive examples and $e_1^-, \ldots, e_n^+-$ are the negative examples) :

`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 :

```
box(a).    box(b).
colour(red).  colour(blue).
has_colour(a, red). has_colour(b, blue).
```

With mode declarations

```
modeh(on_floor(+box)).
modeb(has_colour(+box, #colour)).
modeb(not has_colour(+box, #colour)).
```

And examples E^+ = `{on_floor(a)}` and E^- = `{on_floor(b)}`.

From these examples, the goal statement is generated.

```
goal :- on_floor(a), not on_floor(b).
:- not goal.
```

The mode declaration generated the skeleton rules

```
on_floor(A) :- box(A), rule(1).
on_floor(A) :- box(A), has_colour(A, C1), rule(2, C1).
on_floor(A) :- box(A), not has_colour(A, C1), rule(3, C1).
```

These parts, combined with a choice rule and optimisation statement give the overall encoding:

```
has_colour(a, red).
has_colour(b, blue).
box(a).
box(b).
colour(red).
colour(blue).

goal :- on_floor(a), not on_floor(b).
:- not goal.

on_floor(A) :- box(A), rule(1).
on_floor(A) :- box(A), has_colour(A, C1), rule(2, C1).
on_floor(A) :- box(A), not has_colour(A, C1), rule(3, C1).

{rule(1), rule(2, red), rule(2, blue), rule(3, red), rule(3, blue)}.

#minimise[rule(1) = 1, rule(2, red) = 2, rule(2, blue) = 2, rule(3, red) = 2,
   rule(3, blue) = 2].
```

When ran (using an ASP solver), the Answer Set of this program contains the atom `rule(2, red)`. This rule corresponds to the learned skeleton rule `on_floor(A):- box(A), has_colour(A, red)`, which is stating that a box is on the floor if it is red, which is consistent with the examples.

2.2.5 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 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.

2.3 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.

2.3.1 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 $l \rightarrow r \Leftarrow v_1 = u_1, \ldots, v_n = u_n$, 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.

2.3.2 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.

2.4 The Target Language, Haskell

Haskell is a purely functional programming language, featuring a strongly typed static type system and supporting language features such as pattern matching, list comprehension, higher order functions and monads. This this sections I will detail the Haskell language features relevant to this project.

Functional programming is a programming paradigm that considers the evaluation of a computational function in the same way as a mathematical one. Functional programs are stateless, depending purely on the arguments to the function. By removing side-effects, it can be proven that a call to some function `f` will always return the same result as long as the input arguments are the same. Functional programming is also very similar semantically to logic programming, which makes using a form of programming like ASP to learn it more appealing over other paradigms.

2.4.1 The Haskell Syntax

To illustrate the syntax of Haskell programs, I will give an example of a function written in Haskell.

```
factorial n
  | n == 0 = 1
  | otherwise = n * factorial (n - 1)
```

The first line of the file defines the function name and any arguments it takes. In the above example, there is one argument, `n`.

Then, every other line starts with a "guard" statement. This statement defines when the expression after the `=` should be executed. In the example, the guard statement `| n == 0` holds when the input is 0, and the `| otherwise` statement is a catch-all for all other cases.

The other important syntax feature is `where` statements, which allows for variable substitution. For example, the program

```
f x = x * y
       where y = 5
```

This program simply multiplies by 5, as the variable `y` has value 5 and is substituted in.

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.

I chose these terms as I felt they were expressive enough to be able to represent sufficiently complicated test examples, while not overcomplicating my first attempt at writing the interpreter.

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(x0;x1;x2;x3).

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. Derivation / Expression tree pictures, with eval-with and value-with branches.

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 Generating Skeleton Rules

Initially, it was difficult to decide how to generate the skeleton rules. While it would not be difficult to naively iterate over all combinations up to a certain depth, this method is very inefficient, especially when considering more than one input argument. This is because a sizeable subset of the generated rules are redundant, having semantically equivalent bodies to other rules. For example, the rule bodies `add(x, x)` and `mul(x, 2)` compute the same result, but are counted as distinct and separate skeleton rules.

To reduce the number of redundant rules, I have implemented a number of optimisations. By generating rules in a recursive manner, deeper rules are generated based off shallower ones. Then, by removing simple redundant rules when the depth is small (where they are easier to detect), rules with the same redundancy are removed from deeper iterations.

Another optimisation is the implementation of `where` rules. These rules limit the depth of each individual rule body to one operation deep, making enumerating over all possibilities much easier, and reducing the overall scaling of the number of rules from multiplicative to additive.

4.2.2 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;40),
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.

4.2.3 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.

4.2.4 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(1, N)=N, choose(1, 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, (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, (3, 6)), 3).
example(call(gcd, (9, 6)), 3).

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

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

This tool implements an iterate approach to learning. Start by entering some examples that may synthesise your desired output. Then, examine the returned Haskell and iteratively add more examples, fix incorrect ones and re-learn until you are happy with the result.

6.1.1 Entering examples

In order to use this tool, you first have to enter examples to learn, which specify how the target program behaves on specific inputs.

You can change the number of input arguments by editing the "No. Args" box. While there are no restrictions on the possible arguments, be aware that any increase in number of arguments can greatly affect learning performance. Currently, the only supported argument type is Integer.

6.1.2 The Learning Step

After entering your examples, you can perform learning by clicking the "Run ASP" button.

After a short computation step, the learned haskell is displayed, and there are a few options on how to proceed :

- If the learned program is incorrect, then you can freely add more examples which specify the missing behaviour.
- If you are unsure about the correctness of the program, you can simply test some more complicated answers using example autocompletion.
- If you are happy with the learned program, you can download the generated code, or save it to be re-used by other tasks.

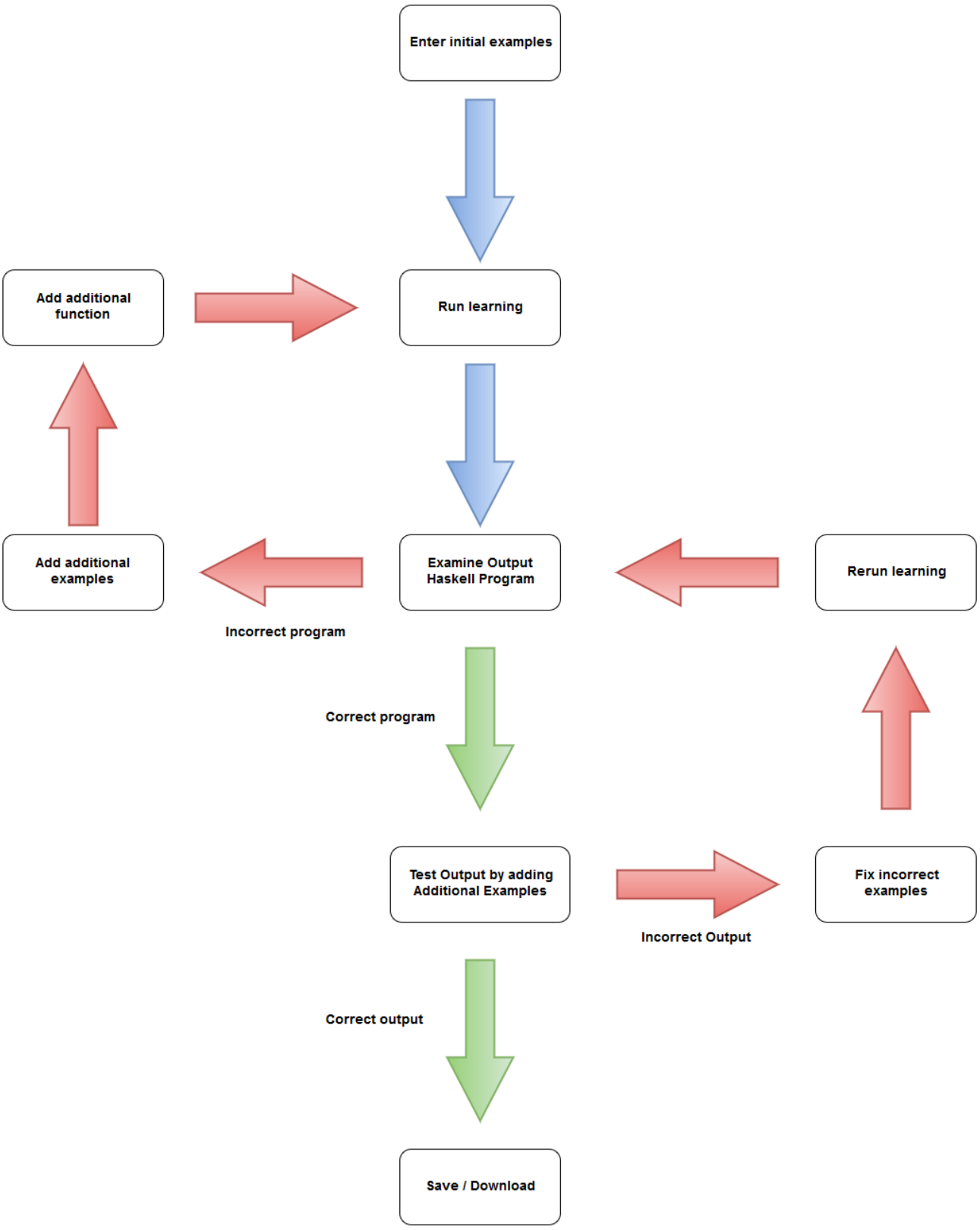

Figure 6.1: The iterative learning workflow

6.1.3 Example autocompletion

If you want to test more complicated values, you can provide examples with valid input, but no output. After learning, these outputs will be completed by the tool, and by analysing

the result compared to the expected value, you can discern the correctness of the learned program.

If you are unsure about program correctness, you can add more examples to be autocompleted which will then be ran on the learned program without a new learning task being started.

6.1.4 Combining functions

Once you have successfully learned a function, you may reuse it in another by pressing the save button while having the "Add to Background Knowledge" box checked. Then, reuse this function in another by first adding a new learning task by opening a new tab. Any learning performed on this new task will have knowledge of the initial function and make use of it in a preferential manner.

6.1.5 Restricting the Language Bias

To help increase learning performance, you can restrict the Language Bias, limiting the operations available to be returned in Haskell. If you have some prior knowledge or suspicion that the target program does not use some available operation, then by selecting the relevant check boxes you can tailor the target language specifically to your target domain.

It is also possible to limit the learning to only tail recursive programs. This feature can offer a large performance increase but does only learn a small subset of programs.

6.1.6 Handling errors

Sometimes, it may not be possible to learn a program from a set of examples. They may be contradictory, or the tool may not be robust enough. In this case, you have a few options open to you :

- Try to remove certain examples to find out which one might be causing a contradiction.
- Separate your function into multiple smaller ones. For example, a Merge Sort program could be separated into smaller splitting and comparision programs.

6.2 The Design Process

Designing the user interface was not an easy process. While I knew the basic features of my UI, to be an iterative learning tool which takes examples as input and displays a learned Haskell program as output, I did not have much of an idea of how to display this to the user. I chose to represent the input fields as a table or spreadsheet as it seemed like an interface that is commonly known and easy to understand. The concept of adding rows and columns, and having functions that work over those columns is one common to spreadsheet tools like Excel, and I thought this would help illustrate the concept of my tool to new users unfamiliar with IFP.

One UI approach I considered was writing the tool as a plugin for an existing spreadsheet program such as Excel or LibreOffice Calc. While this would have saved a majority of the frontend design and implementation work, it would have meant that my tool has to rely on user's knowledge of the host program, and limiting what features I could add.

Now that I had chosen a spreadsheet approach, I had to design how it would look. Initially using paper sketches and online formatting tools, I started building a picture of what the UI would look like. However, my ideas started looking more like a table than a spreadsheet, as I would not need all the functionality a typical spreadsheet provides. In addition, Javascript libraries for making spreadsheets are quite hard to find and implement, while ones with table functionality are common, as tables are built-in HTML tag.

6.3 Writing the Backend

The first part of my tool that needed a backend was the part which generates skeleton rules. Instead of finding some way in ASP to enumerate over all possible rule bodies, it seemed easier to write a Java program to perform this generation. I chose Java as it is a language I am very familiar with, reducing the number of new technologies I would have to learn to start implementation. In addition, Java can be quite easy to maintain as the size of the project grows, although it can be overly verbose at times.

After handling skeleton rule generation, the backend also needed to handle other tasks that could not be performed using ASP.

- Converting from ground `choose` atoms as produced in the answer set to full choice rules.
- Generating Haskell from the chosen ASP rules.
- Running the Haskell program to auto-complete any examples.
- Managing when to just run Haskell and when to re-learn.

While it might seem that converting from ASP to valid Haskell would be difficult, it turned out to be a surprisingly easy process due to the chosen subset of the target language and way I have defined rule bodies. Because there are only arithmetic operations with two arguments, the tool can use recursively iterate through the expression, building up the Haskell string as it runs using simple string formatting. It is equally easy to generate guard statements from the bodies of chosen `match` rules

6.4 Technologies Used

To make use of the Java backend, I needed a UI framework which could easily integrate with it, while also not being too complicated to learn. In addition, I wanted a way to implement a web based UI instead of a traditional Desktop application, as it would allow the tool to be hosted on the cloud, available to anyone with a web browser.

6.4.1 Play Framework and Akka

Because I wanted to build a web app while having a Java backend, it made most sense to use the Play Framework to link the two together. The Play Framework provides a way to easily integrate a HTML and Javascript frontend with a Java backend, through HTTP requests. Methods on the backend that need to be accessed on the frontend can be written into a Controller class, whose methods become exposed as `GET`, `POST` and `DELETE` endpoints. These endpoints are then accessable in the frontend, through ajax calls written in the Javascript.

Play also allows for processes to be ran in the background, asynchronously. This was useful for the tool, as I did not want it to be unresponsive while waiting for the potentially long time the learning task takes. To achieve this, I made use of Akka actors. Akka is a toolkit used to build concurrent systems made up from actors, abstract concurrent processes. As part of my backend, I define the learning task as one of these actors, which is initiated when the frontend submits a new task. Then, the frontend repeatedly queries the backend until the learning is complete, at which point the learned Haskell is returned.

6.4.2 Bootstrap

For the user interface, I wanted a quick, simple way to design it, looking clean and structured without anything overly complicated. To achieve this, I used Bootstrap, a simple css and javascript library, containing a large built in collection of common HTML elements such as toolbars, modals, tables and tab. Having used Bootstrap in projects before, it was familiar enough that it made writing my initial UI frontend quick and painless, and allowed me to easily make changes based on user feedback.

6.5 User Feedback and Evaluation

To help with the design of the user interface, it was important to get feedback from users wherever possible. This feedback started with friends and supervisors, and was given informally as we discussed each stage of the UI. With friends, we frequently worked together and at such an early stage it is easy to ask for feedback and iterate based on this.

As the UI came nearer to completion, I realised that more formal feedback was necessary. By getting in touch with younger students and family with a non-technical background, I could get more unbiased and descriptive feedback from a variety of perspectives. This feedback helped fix bugs, add extra features, gain new examples and uncover limitations in the learning.

Chapter 7

Critical Evaluation

In this chapter, I describe the set of experiments used to show the increase in performance from the implementation of the constraint-based approach.

I have measured the performance by running both versions of the learner on five target problems, and returning both the generated code and the time taken to learn, as returned by clingo. The problems are defined by a set of Input / Output examples which cover enough cases to correctly learn the target function. Solutions are defined as a subset of the generated Answer Set, containing only `choose`, `choose_match` and `choose_where` atoms, representing chosen rules from the skeleton rules.

The five examples I am running are defined as follows.

- **Factorial** : The recursive definition of the mathematical factorial operation, denoted by `n!` . It is defined as the product of all positive integers less than or equal to `n`.
- **Fibonacci** : The Fibonacci numbers belong to the following sequence - `1, 1, 2, 3, 5, 8, 13`, where each member of the sequence is equal to the sum of the previous two numbers.
- **Powers of Two** : A simple function which recursively defines the calculation of 2^n.
- **Tail Recursive Factorial** : The tail-recursive definition of the factorial function. Takes two arguments, where the second is an accumulator in which the product is calculated.
- **Greatest Common Divisor** : A version of Euler's algorithm to calculate the greatest common divisor of two numbers.

All examples have been run on –LAB PC SPECS–

7.1 Generated Code

7.1.1 Factorial

Input Examples

```
example(call(fac, 0), 1).
example(call(fac, 1), 1).
example(call(fac, 2), 2).
example(call(fac, 3), 6).
```

Interpreted approach results

```
fac x
  | x == 0 = 1
  | otherwise = x * x0
  where x0 = f x1
        where x1 = x - 1
```

Constraint approach results

```
fac x
  | x == 0 = 1
  | otherwise = (fac (x - 1)) * x
```

7.1.2 Fibonacci

Input Examples

```
example(call(f, 1), 1).
example(call(f, 2), 1).
example(call(f, 3), 2).
example(call(f, 4), 3).
example(call(f, 5), 5).
example(call(f, 6), 8).
```

Interpreted approach results

```
fib x
  | x == 1 = x
  | x == 2 = x - 1
  | otherwise = x0 + x2
  where x0 = fib x1
        where x1 = x - 1
              where x2 = fib x3
                    where x3 = x - 2
```

Constraint approach results

```
UNSATISFIABLE
```

7.1.3 Powers of 2

Input Examples

```
example(call(f, 0), 1).
example(call(f, 1), 2).
example(call(f, 2), 4).
example(call(f, 3), 8).
```

Interpreted approach results

```
power2 x
  | x == 0 = x + 1
  | otherwise = x1 + x2
        where x1 = f x0
                where x2 = f x0
                        where x0 = x - 1
```

Constraint approach results

```
power2 x
  | x == 1 = 1
  | otherwise = (f (x - 1)) * 2
```

7.1.4 Tail Recursive Factorial

Input Examples

```
example(call(f, (0, 1)), 1).
example(call(f, (1, 1)), 1).
example(call(f, (2, 1)), 2).
example(call(f, (3, 1)), 6).
example(call(f, (2, 3)), 6).
```

Interpreted approach results

```
fac x y
  | x == 0 = y
  | otherwise = fac x0 x1
  where x0 = x - 1
        where x1 = x * y
```

Constraint approach results

```
fac x y
  | x == 0 = y
  | otherwise = fac (x - 1) (x * y)
```

7.1.5 Greatest Common Divisor

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

Interpreted approach results

```
gcd x y
  | y == 0 = x
  | x > y = gcd y x0
  | otherwise = gcd x1 x
  where x0 = x - y
        where x1 = y - x
```

Constraint approach results

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

7.2 Performance Statistics

The results from these experiments tend to show the result that apart from in the simple case, the constraint based approach performs around 50% better than the interpreted approach across all examples. In the simple factorial example, the interpreted approach was quicker due to the small set of skeleton rules effecting performance in relatively small way.

However, the constraint based approach is less expressive, as highlighted by the fibonacci example. The fibonacci program consists of two recursive calls, which cannot be learned by this approach as it becomes difficult to maintain the equality constraints. For example, consider the equality constraint `eq(add(call(f, (x - 1)), call(f, (x - 2))), 4)`. The tool cannot continue evaluating this constraint as it does not know the value of either `call` expressions.

Table 7.1: Approach Performance Comparison

Performance		
Function	Interpreted	Constraint Based
Factorial	Time : 1.640 Prepare : 0.060 Prepro. : 0.020 Solving : 1.560	Time : 1.240 Prepare : 0.200 Prepro. : 0.140 Solving : 0.900
Fibonacci	Time : 807.240 Prepare : 0.060 Prepro. : 0.020 Solving : 807.160	Unsatisfiable
Power of Two	Time : 35.840 Prepare : 0.040 Prepro. : 0.040 Solving : 35.760	Time : 5.580 Prepare : 0.380 Prepro. : 0.340 Solving : 4.860
Tail Recursive Factorial	Time : 419.860 Prepare : 2.060 Prepro. : 0.840 Solving : 416.960	Time : 173.900 Prepare : 5.540 Prepro. : 1.440 Solving : 166.920
GCD	Time : 1546.820 Prepare : 1.940 Prepro. : 0.980 Solving : 1543.900	Time : 1599.420 Prepare : 11.720 Prepro. : 2.100 Solving : 1585.600

7.3 Comparison to Existing Tools

In this section, I will detail how other existing inductive functional programming (IFP) tools perform in comparison to mine, highlighting any benefits or restrictions to both mine and the existing tools. More information on these tools can be found in the background section.

For evaluation, I have ran both systems on the examples detailed in the section above, showing a comparison between code generated and running time between both existing systems and my tool's constraint based approach.

7.3.1 MagicHaskeller

MagicHaskeller is an IFP system that is based on the generate-and-test approach to IFP. It works by systematically generating a stream of gradually more complicated functions (based on length), and keeping the ones which cover the input examples. The system is made more efficient through smart restrictions of target programs based on the type of the input examples, and through heavy use of in built functions, both linear and higher order.

Input to MagicHaskeller is any Haskell expression which returns a boolean value, as long as it does not contain any `let` or `while` clauses. For example, to learn the factorial program, you can give as input the examples `\f -> ( f 0 == 1 )&& ( f 1 == 1 )&& ( f 2 == 2 )&& ( f 3 == 6 )`. Here, each positive example is concatenated with `&&` to produce the overall input.

Full tables of generated programs and performance timings can be found respectively in tables 7.2 and 7.3

What is notable from these results is that MagicHaskeller fails to learn direct recursion, relying instead on its large library of in-built functions. This is most visible on the GCD and Fibonacci examples, where MagicHaskeller can simply make use of Haskell's inbuilt gcd function, but cannot learn the purely recursive Fibonacci. This is because of a deliberate choice by the creator of MagicHaskeller, as removing direct recursion decreases the complexity of the search space.

7.3.2 Igor II

Igor II is an IFP system that is designed to learn purely recursive programs, and is different to my tool in important ways.

- Igor II does not have any built in background knowledge, which means that any functions to be learned need all basic operations and constants defined at the same time as the examples. This is only a small restriction for learning lists because the set of constants and operations to be defined are limited to the empty list and list prepending respectively. However, when learning arithmetic functions it is very complicated to define addition and multiplication in terms of the basic succession operation.
- The output from Igor II is a list of hypotheses, rather than a program or lambda function. While this increases the expressivity, it severely reduces readability and usability by users less experienced in the field.

Table 7.2: MagicHaskeller Generated Code Comparison

Generated Code		
Function	My Solution	MagicHaskeller Solution

Factorial	`f x`
	`  | x == 0 = 1`
	`  | otherwise = x * f(x - 1)`
		`f =`
		`  (\a -> product [1..a])`

Tail Recursive Factorial	`f x n`
	`  | x == 0 = n`
	`  | otherwise = f (x - 1) (x * n)`
		`f =`
		`  (\a b -> product [b..a])`

GCD	`gcd x y`
	`  | x == y = x`
	`  | x > y = gcd x - y y`
	`  | x < y = gcd y x`
		`f = (flip gcd)`

Fibonacci	`fib x`
	`  | x == 1 = x`
	`  | x == 2 = x - 1`
	`  | otherwise = x0 + x2`
	`  where x0 = fib x1`
	`  where x1 = x - 1`
	`  where x2 = fib x3`
	`  where x3 = x - 2`
		No Result!

Table 7.3: MagicHaskeller Performance Comparison

Performance		
Function	My Performance	MagicHaskeller
Factorial		
Tail Recursive Factorial		
GCD		
Fibonacci		

Chapter 8

Conclusions and Future Work

In this chapter I will give an overview of the went well during development of the tool , and an list of features that would have been implemented in the learning tool given more time.

8.1 Conclusions

Overall, I am very pleased with the finished learner. While I did not achieve everything I set out to accomplish, I was deliberately ambitious with those goals and to achieve it all would have been overly difficult. In addition, the tool has been designed in such a way that adding these additional features would not be overly difficult and require a large amount of extra code.

8.1.1 Learning

I am most pleased with the current version of my learning task. It details a successful new approach to Inductive Functional Programming, showing that it is possible to use ASP to learn directly recursive programs in an efficient way.

The constraint based approach exceeded expectations, allowing a performance increase of approximately 50%. This performance increase allows the tool to attempt learning more expressive programs than the interpreted approach, while not sacrificing performance.

Whilst the fact that I had to implement an entire new learning approach towards the end of the project has meant that the size of the target language is smaller than planned, it would only be a matter of time before additional features would be added, rather than them not being possible to add.

8.1.2 UI

I am also proud of the quality of the user interface I created. Usage of the bootstrap libraries has meant that the UI looks clean and functional without being overly complicated. As an added bonus, it was not difficult to implement, allowing me to easily change and modify the UI based on feedback.

My usage of the Play Framework has also increased the usability of the tool. The learning running as a asynchronous process means that the interface feels fluid, allowing the user to change tabs and work on other functions while waiting for learning to be complete. Play

has also made development of the UI easier, clearly separating the frontend and backend and keeping my code organised and fairly simple.

One feature that works well is the generation of Haskell programs from ASP. While the output from clingo is simply an answer set containing atoms specifying the chosen rules, this is converted into valid Haskell by the UI. The UI also allows for the Haskell to be downloaded and ran locally by the user, ran by the UI itself to check the validity of the learned program. This feature makes my tool stand out from other existing systems, which have output that can be very difficult to analyse or convert into something runnable.

8.2 Future Work

8.2.1 Type Usage

One of the main features the current version of the learning tool is missing is the ability to handle input and return types that are not integers. This adds a significant restriction on what the tool can learn, and if implemented would not adversely affect performance.

The implementation of this feature would be based on performing analysis on the types of the examples. By using the UI to calculate the type of each input argument and the type of the example, it would be possible to evaluate which operations can be performed on those input types to return the correct output. It would then be possible to generate the set of skeleton rules based on these operations.

As an example, consider the `isPrime` function, which returns a true if the input to the function is a prime number, and false if not. It is important to know that the function takes two arguments - the first being the number to check the primality of, and the second being a divisor which is checked against the first argument and decremented at each recursive step.

Analysing the types of this example would tell the tool that all of the input types are integer, and that the output type is boolean. This would limit the tool to only generate skeleton rules which have bodies which return booleans - i.e having `==`, `>`, `<` or the recursive call to `isPrime` in the top level in the body expression.

This type of implementation would not affect performance because it is not increasing the size of the skeleton rules, only changing their content. In some cases even, the number of skeleton rules would be fewer if restricted in this way, as there may be fewer in-built operations to enumerate over.

8.2.2 Lists and Strings

Another key part of the Haskell language I am missing in my potential target language is list functionality. Without lists, a large feature of functional programming is missing, reducing the overall expressiveness of the possible learning domain.

Implementing the learning of lists would not be a trivial task, although it is already partially complete. The ASP interpreter is already expressive enough to handle lists, representing them as nested tuples. In a similar way to how the Haskell list `[0, 1, 2, 3]` is internally

represented as multiple items prepended together, `(0 : (1 : (2 : (3 : []))))`, the list is represented in ASP as `(0, (1, (2, (3, e))))`, where `e` is an internal representation of the empty list.

The interpreter handles these lists in the same way it handles multiple arguments. If it has to evaluate a tuple, it then generates `eval` terms for call complex arguments. Similar behaviour holds for values of expressions and checking for complexity. This behaviour is covered by the following rules :

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

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

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

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

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

Implementing lists for the constraint based approach would be similar. Through use of rules which define equality on tuples, it should not be difficult to maintain the equality constraints which fail when a contradiction occurs. The check for termination would work in a similar way, although some advanced implementation of match rules may be necessary to check for list patterns.

The difficult part of lists would be enumerating all of the different possibilities. While my current implementation allows for a small range of constants to be learned as part of the skeleton rules, to allow for lists this range would have to be a lot wider. and consist of all combinations of constants up to a certain list length.

The implementation on string operations would work in a similar way, by treating strings as list of characters. The string `"hello"` would be represented in ASP as the list `(h, (e, (l, (l, (o, e)))))`. This would allow all basic operations on lists to be performed on strings, and allow them to be learned in the same way.

8.2.3 Expanding the Background Knowledge

To increase the expressiveness of the learned functions, it would be beneficial to increase the number of inbuilt functions in the background knowledge, to better represent functions a person may write. The functions I have considered for addition are part of the `Prelude.hs` standard Haskell library

List Processing	Arithmetic	Tuples
`length`	`div`	`zip`
`++`	`mod`	`unzip`
`reverse`	`gcd`	`fst`
`head`	`even`	`snd`
`last`	`odd`	
`tail`	`sum`	
	`product`	

Usage of these functions would be implemented by adding more rules to the skeleton rules. The skeleton rules would contain all possible usages of these in-built functions with the given arguments, restricted by the types of the examples. Then, the ASP interpreter would have to contain rules which handle the generating the value of expressions containing these functions. Through use of addition lua scripting, the complexity of this task could be reduced, allowing for example the calculation of a length of a list to be evaluated in one rule instead of many.

In addition to these in-built functions, implementing higher-order functions would also increase the expressiveness of learned functions. The functions `map`, `fold` and `filter` would be added to the skeleton rules in a similar way, with the addition of having to enumerate all possible functions as additional arguments. However, evaluating the higher order functions might prove to be more difficult. Applying operations to every element in a list would be difficult to evaluate using just ASP, but through use of lua scripting, again it would be possible, albeit slow.

8.2.4 Advanced Skeleton Rule Generation

One of the main advantages to existing Inductive Functional Programming implementations is the way they generate their search space. MagicHaskeller generates a stream of valid functions, ordered by depth, and checks each one sequentially against the examples. If my learning tool were to implement such a system then it would have the potential to greatly increase performance. Currently, my skeleton rule generation is a naive process, containing many irrelevant rules which could not return valid programs, and my implementation of optimisation means that clingo has to enumerate all possible programs before returning a result, instead of being able to return as soon as a solution is found as would occur using a stream of rules. However, it is not clear how this would be easily implemented in ASP.

Another potential skeleton rule generation optimisation would be to be explicit with the difference between rules that can appear in the base case and the recursive cases of generated functions. Currently, all possible rules can appear in the base case of a recursive function, even complex function calls. This means that the learning task has to evaluate a large number of redundant possibilities, reducing performance.

To avoid this, the tool would limit the possible rules for base cases to simple arithmetic functions that are only one, potentially two operations deep. Then, the recursive cases would be limited to only expressions which contain calls to functions or in-built operations. Theoretically, this optimisation could lead to around a 50% increase in performance on functions with two cases, one base case and one recursive case, as it would greatly reduce the time taken to enumerate the base case, which in this case is half the rules needed to be learned.

8.2.5 Supporting Multiple Function Calls

The main limitation on the constraint based learning approach is that it cannot handle multiple function calls. Usually when evaluating equality constraints, one of the two arguments is a constant, allowing the tool to calculate the value of any non-constant arguments. For example, the evaluation of constraint `eq(mul(3, call(f, 2)), 6)` occurs with the operations

$$\begin{aligned} 3 * f(2) &= 6 \\ f(2) &= 6/3 \\ f(2) &= 2 \end{aligned}$$

However, if the tool encounters a equality constraint with a function call as both arguments, similar to `eq(add(f(6), f(5)), 8)`, then it cannot proceed as it does not know the constant value of either function call.

Ways to overcome this problem involve making assumptions about the value of either function call. As the value of constants in the tool is constrained to a small subset of integers, the range of possible values a function call can return is also constrained to this subset. Using this information, it would be possible to make assumptions about the value of either function call in the equality constraint, through use of choice rules.

```
1 { value(call(f, 5), R) : const_number(R) } 1.
1 { value(call(f, 6), R) : const_number(R) } 1.
```

This rule generates a `value(call(f, 5), R)` atom for each integer constant, then chooses each one to be tried in evaluation of the equality constraint. Using the example above, if the tool chooses `value(call(f, 5), 3)` then it can now evaluate the constraint as follows

$$\begin{aligned} f(6) + f(5) &= 8 \\ f(6) &= 8 - 3 \\ f(6) &= 5 \end{aligned}$$

However, for this to allow the constraint to hold, more restrictions are required. Both sides of the constraint have to be explored, as it could be possible for an equality constraint to hold on one side but not the other. In addition, all `value` atoms generated have to be consistent. If for one constraint the tool chooses that `value(call(f, 5), 3)` and another chooses `value(call(f, 5), 5)`, then this should be unsatisfiable, as the functional nature of the target language means that a function can only return one result.

The main downside of this approach would be that it has the potential to severely reduce performance. Any additional choice rules increase the complexity of the learning, adding more combinations of rules to be learned. Perhaps it is good enough to accept that the trade off to the increased performance of the constraint based approach is reduced expressivity.

8.2.6 Parallel Learning

One way to handle having multiple approaches would be to not completely replace the initial approach, and instead run them both in parallel. Through the user interface, when a new

learning task is started it would be possible to start clingo runs of both approaches. Then, whichever run returns first is displayed as the learned program. While this would usually be the constraint based approach, occasionally it will fail as that approach is less expressive. This restriction does not apply to the interpreted approach, which would take longer but could find solutions the first approach misses.

This would be implemented as part of the UI through use of Akka actors. As my UI already makes use of these to allow the learning to happen in the background, it would not be difficult to run another actor, then terminate both once one returns a result.

Appendix A

Full Experimental Results

A.1 One Argument Programs

A.1.1 Factorial

Input Examples

```
example(call(fac, 0), 1).
example(call(fac, 1), 1).
example(call(fac, 2), 2).
example(call(fac, 3), 6).
```

Interpreted approach results

```
choose_match(1,1,0)
choose_match(2,2)
choose(1,3,1)
choose(2,9)
choose_where(34)
choose_where(45,1)

Models : 1
Optimization: 2 3
Time : 1.640
  Prepare : 0.060
  Prepro. : 0.020
  Solving : 1.560

fac x
  | x == 0 = 1
  | otherwise = x * x0
  where x0 = f x1
        where x1 = x - 1
```

Constraint approach results

```
choose_match(1,1,0)
choose_match(2,2)
choose(1,1,1)
choose(2,37,1)

Models : 1
Optimization: 1
Time : 1.240
  Prepare : 0.200
  Prepro. : 0.140
  Solving : 0.900

fac x
  | x == 0 = 1
  | otherwise = (fac (x - 1)) * x
```

A.1.2 Fibonacci

Input Examples

```
example(call(f, 1), 1).
example(call(f, 2), 1).
example(call(f, 3), 2).
example(call(f, 4), 3).
example(call(f, 5), 5).
example(call(f, 6), 8).
```

Interpreted approach results

```
choose_match(1,1,1)
choose_match(2,1,2)
choose_match(3,2)
choose(1,1)
choose(2,13,1)
choose(3,84)
choose_where(30,1)
choose_where(49)
choose_where(83)
choose_where(75,2)

Models : 1
Optimization: 4 1
Time : 807.240
  Prepare : 0.060
  Prepro. : 0.020
  Solving : 807.160

fib x
  | x == 1 = x
  | x == 2 = x - 1
  | otherwise = x0 + x2
  where x0 = fib x1
        where x1 = x - 1
              where x2 = fib x3
                    where x3 = x - 2
```

Constraint approach results

```
UNSATISFIABLE

Models : 0
Time : 0.020
  Prepare : 0.020
  Prepro. : 0.000
  Solving : 0.000
```

A.1.3 Powers of 2

Input Examples

```
example(call(f, 0), 1).
example(call(f, 1), 2).
example(call(f, 2), 4).
example(call(f, 3), 8).
```

Interpreted approach results

```
choose_match(1,1,0)
choose_match(2,2)
choose(1,3,1)
choose(2,84)
choose_where(30,1)
choose_where(49)
choose_where(64)

Models : 1
Optimization: 3 3
Time : 35.840
  Prepare : 0.040
  Prepro. : 0.040
  Solving : 35.760

power2 x
  | x == 0 = x + 1
  | otherwise = x1 + x2
       where x1 = f x0
              where x2 = f x0
                     where x0 = x - 1
```

Constraint approach results

```
choose_match(1,1,0)
choose_match(2,2)
choose(1,1,1)
choose(2,38,1,2)

Models : 1
Optimization: 1
Time : 5.580
  Prepare : 0.380
  Prepro. : 0.340
  Solving : 4.860

power2 x
  | x == 1 = 1
  | otherwise = (f (x - 1)) * 2
```

A.2 Two Argument Programs

A.2.1 Tail Recursive Factorial

Input Examples

```
example(call(f, (0, 1)), 1).
example(call(f, (1, 1)), 1).
example(call(f, (2, 1)), 2).
example(call(f, (3, 1)), 6).
example(call(f, (2, 3)), 6).
```

Interpreted approach results

```
choose_match(1,1,0)
choose_match(2,7)
choose(1,2)
choose(2,27)
choose_where(51,1)
choose_where(69)

Models : 1
Optimization: 2 2
Time : 419.860
  Prepare : 2.060
  Prepro. : 0.840
  Solving : 416.960

fac x y
  | x == 0 = y
  | otherwise = fac x0 x1
  where x0 = x - 1
        where x1 = x * y
```

Constraint approach results

```
choose_match(1,2,0)
choose_match(2,1)
choose(1,3)
choose(2,184,1)

Models : 1
Optimization: 3
Time : 173.900
  Prepare : 5.540
  Prepro. : 1.440
  Solving : 166.920

fac x y
  | x == 0 = y
  | otherwise = fac (x - 1) (x * y)
```

A.2.2 Greatest Common Divisor

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

Interpreted approach results

```
choose_match(1,2,0)
choose_match(2,6)
choose_match(3,7)
choose(1,1)
choose(2,30)
choose(3,81)
choose_where(53)
choose_where(77)

Models : 1
Optimization: 1 2
Time : 1546.820
  Prepare : 1.940
  Prepro. : 0.980
  Solving : 1543.900

gcd x y
  | y == 0 = x
  | x > y = gcd y x0
  | otherwise = gcd x1 x
  where x0 = x - y
        where x1 = y - x
```

Constraint approach results

```
choose_match(1,3)
choose_match(2,4)
choose_match(3,6)
choose(1,2)
choose(2,150)
choose(3,190)

Models : 1
Optimization: 2
Time : 1599.420
  Prepare : 11.720
  Prepro. : 2.100
  Solving : 1585.600

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

References

[1] K. Clark. *Readings in nonmonotonic reasoning*, chapter Negation as failure, pages 311–325. Morgan Kaufmann Publishers, 1987.