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.

All examples have been run on –LAB PC SPECS–

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.

7.1 One Argument Programs

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

```
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
```

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

```
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
```

7.2 Two Argument Programs

7.2.1 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)
```

7.2.2 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
```

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 7.1 and 7.2

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

Table 7.1: 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 7.2: 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 features that would have been implemented in the learning tool given more time, and some opinions on the positives and negatives of my implementation.

8.1 Future Work

8.1.1 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.

8.1.2 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.

8.1.3 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.

8.1.4 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.

8.1.5 Supporting Multiple Function Calls

8.1.6 Loading known functions

8.1.7 Parallel Learning

8.2 Conclusions

8.2.1 What worked

8.2.2 Areas for improvement