# Synthesis of Simple While Programs Using Answer Set Programming

Joseph Crowe
Supervised by Krysia Broda and Mark Law

18th June 2015

**Abstract**

We address the problem of generating programs in a simple imperative programming language supporting integer arithmetic, from a high-level specification of the program's behaviour. We focus in particular on users lacking programming experience, and so allow the specification take the form of a list of input/output examples of expected program behaviour.

Using Answer Set Programming and inductive learning techniques, it is seen that this is enough to generate short programs, and we present a tool for automating this process.

This tool is extended with the ability to take a functional specification of the program's behaviour consisting of a logical precondition and postcondition, and iteratively generate a minimal set of examples needed to find a matching program.

We present details of how this is implemented using Answer Set Programming, and propose promising directions in which further research could be done.

## Acknowledgements

This thesis was written for my final year undergraduate project at Imperial College London. For its completion, I owe my thanks to:

- Krysia Broda and Mark Law, my supervisors, for their advice, support, tolerance, and many valuable suggestions over the course of the project.
- Alessandra Russo for her insightful feedback and suggestions given during the interim project review.
- Duncan Gillies and Barbara Claxton for being very helpful during the final stages of the project.

# Contents

# Chapter 1

# Introduction

One of the aims of computer science is to make it easier for people to generate *computer programs*, which are the machine-readable representations of desired behaviour executed by a computer in order to perform some useful computation.

In many cases the user might not have a clear idea of how a calculation is to be performed, but will be able to point out some concrete examples of its expected behaviour: the user intuitively knows that for certain inputs, they expect the program to behave in a certain way. This is a problem faced by students of Computing, for example, which if addressed could be beneficial to their education.

While certainly not the most reliable way of specifying a computer program, it might be the best approximation that a user can produce with limited knowledge of the problem domain; and if the examples are sufficiently comprehensive, it has been seen that it is often enough to make a good guess at what program the user has in mind.

This project concerns the generation of *imperative* computer programs, consisting of sequences of instructions to be executed in order, which are relatively easy to compile into a form that can be executed on most computing devices without further processing.

In particular, we restrict our attention to a *simple while language* containing the minimal features for computation involving whole numbers: branching, using *if statements*; looping, using *while loops*, and some basic arithmetic operators such as addition and multiplication. See Chapter 4 for a formal description of this language.

An example of such a program is the following, which takes as input an initial value of `x`, say $x_0 \geq 0$, and leaves `y` equal to the $x_0$th power of 2, $y = 2^{x_0}$:

```
y = 1;
while (x > 0) {
   y = y * 2;
   x = x - 1;
}
```

We now have a way to instruct a computer to calculate the powers of two, but is it necessary for a person to write out the program in full? It might be easy in this case, but for more complicated programs this can be time-consuming, error prone, and dependent on specialised knowledge.

Lacking the inclination to write a program in the above form, we might wish instead to specify its behaviour as mathematical relation between its input and output variables, such as $y = 2^{x_0}$; or perhaps we do not even have a mathematical expression in mind, but simply wish to provide a list of examples of what the program should do:

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $y$ | 1 | 2 | 4 | 8 | 16 | 32 |

The objective of this project is to provide automatic programming tools that address this problem by allowing a program to be generated from a list of input/output examples it is desired to satisfy, or from a mathematical or logical relation which is desired to hold over the whole domain of input/output values.

There is a significant body of previous work in the automatic synthesis of programs (see Chapter 2). Most can be classified as *inductive*, which take a machine-learning approach to generalising a set of input/output examples with a program in a particular language; or *deductive*[2][4][8], which seek to construct a program which provably satisfies a high-level representation of desired behaviour.

There have been recent successes in inductive program synthesis[10], which have inspired this project to take an inductive approach to generating imperative programs; however, the problem in its full generality is far from solved, and with the application of new tools and techniques, we hope to break new ground and document their effectiveness.

In particular, we use a type of logic programming called Answer Set Programming (see Chapter 3), which was previously used for inductive synthesis of logic programs, as the main part of a system capable of inductively synthesising imperative programs. For running Answer Set Programs we use *Clingo 3* from the Potsdam Answer Set Solving Collection (Potassco)[11].

## 1.1 Contributions

- We present a tool which, given a list of input/output examples over a given variable domain will generate a program satisfying the examples, if one exists given the certain constraints. See Chapter 5.
- We allow the structure of the generated program to be partially defined by the user, and show that this results in a significant increase in speed of generation. See Chapter 5.
- We extend this system with the ability to judiciously generate examples from a functional specification written in propositional logic, instead of relying on a list of examples from the user. See Chapter 6.

# Chapter 2

# Background: Automatic Program Synthesis

The notion of automatic programming, in the sense of allowing a computer to take part in the writing of a computer program, has its roots in the design of optimising compilers to translate a high-level program into an efficient low-level program while preserving behaviour. The *Fortran I* compiler[1] is a well known early example of this. Since then, researchers have attempted to bridge increasingly wider gaps between specification language and output language. More detailed surveys of the subject can be found at e.g. [3][5][7].

## 2.1 Deductive Program Synthesis

In the *deductive* variety of program synthesis, in which the task is to derive a correct program from a formal specification, there are several documented approaches. The oldest is *transformational synthesis*, which seeks to transform the specification into a correct program through multiple correctness-preserving changes. See [2] for an example.

Also used in deductive synthesis is the methodology of *schema-guided synthesis*, in which templates for program fragments in the target language, called *schemas*, are used to build a larger program. This reduces the complexity of the task, as the work of constructing and proving the correctness of the schema has already been done. For example, a theoretical system is developed at [6], and a more recent application to numerical computation is given at [8].

Finally, the *constructive* approach to deductive program synthesis uses the fact that the specification can be written as a formula whose satisfiability asserts the existence of a program with the specified behaviour; and a proof of the formula's satisfiability is equivalent to the construction of a such a program. Constructive program synthesis therefore borrows the power of theorem-proving techniques to perform the difficult part of program synthesis. An example of this applied to the synthesis of logic programs is given at [4].

More recent advances in deductive program synthesis use combinations of the

above techniques. For example, in [9], it is shown that given a functional specification of an imperative program, relating its inputs and outputs with logical formulae, and given also the looping structure of the the program, it is possible to infer a program satisfying these constraints, using techniques previously used for program verification.

## 2.2 Inductive Program Synthesis

On the other hand, in *inductive* program synthesis, the task is to produce a hypothetical program, given an incomplete specification of its behaviour, which usually takes the form of a list of input/output examples. This can be viewed as a form of machine learning, where the synthesised program is a hypothesis generalising the given examples.

David Perelman, Sumit Gulwani, et al. in 2014 developed a system[10] based on the principle of "programming by example," which generates and iteratively refines a program from a sequence of increasingly more general sets of input/output examples. Notably, their system is not limited to imperative programs, but allows arbitrary domain-specific languages to be defined and used with the system.

# Chapter 3

# Background: Answer Set Programming

Answer Set Programming (ASP) is a form of logic programming particularly suited to search problems over large domains (such as the domain of all imperative programs of a given length). An Answer Set Program distinguishes the result to be computed by a set of propositional facts and rules in the domain. Importantly, ASP includes *aggregates*, *integrity constraints*, and *negation as failure*, where a rule can be conditional on the inability of prove that some proposition is true, which allows *non-monotonic reasoning*. The result of running an ASP is an *answer set* of facts which is consistent with the program, forms a *stable model*, and is minimal in a certain sense. For further details, see [12], which gives a concise introduction to the paradigm.

## 3.1 Previous Work using ASP

In 2013, Domenico Corapi et al. developed ASPAL[13], a system written in ASP for solving classical Inductive Logic Programming tasks, which brings to bear against these problems the efficiency of existing ASP solvers, and allows their optimisation features to be used to direct the search towards a desired optimal solution.

Building on this, in 2014, Mark Law et al. in 2014 developed ILASP[14], implementing a form of ILP where an Answer Set Program representing the solution is learned inductively from positive and negative examples, with applications in agent planning.

# Chapter 4

# Programs in the While Language

## 4.1 Syntax and Semantics

The output language of our system is an imperative programming language that can be described as a "simple while language". A running program's state is kept in a fixed number of global integer variables, and the execution path is controlled by *if* and *while* statements. This language is therefore Turing complete, its expressive power in principle no less than any other programming language.

### 4.1.1 Syntax

The following context-free grammar gives the (abstract) syntax of the While language. The basic elements of a program are integer constants and named variables.

$$\begin{aligned} \text{Const} &\to ... \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid ... \\ \text{Var} &\to a \mid b \mid c \mid ... \mid x \mid y \mid z \mid ... \end{aligned}$$

A While program is a possibly empty sequence of *commands*, where a *command* assigns a value to a variable, or takes the form of an `if` or `while` statement with a boolean guard and a further sequence of commands:

$$\begin{aligned} \text{Prog} &\to \text{Cmds} \\ \text{Cmds} &\to \varepsilon \mid \text{Cmd}\texttt{;}\ \text{Cmds} \\ \text{Cmd} &\to \text{Var}\ \texttt{=}\ \text{Expr} \\ &\quad \mid \texttt{if (}\text{Bool}\texttt{) \{}\ \text{Cmds}\ \texttt{\}} \\ &\quad \mid \texttt{while (}\text{Bool}\texttt{) \{}\ \text{Cmds}\ \texttt{\}} \end{aligned}$$

An arithmetic expression is a sum, difference, product, integer quotient, or integer remainder of two constants and/or variables. A boolean guard can test

the relation less-than ($<$) or less-than-or-equal ($\leq$) on constants and variables. The structure of expressions is deliberately kept as simple as possible, to reduce the search space of programs:

$$\begin{aligned} \text{Expr} &\rightarrow \text{LExpr} \ (\ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}\ )\ \text{LExpr} \\ \text{Bool} &\rightarrow \text{LExpr} \ (\ \texttt{<} \mid \texttt{<=}\ )\ \text{LExpr} \\ \text{LExpr} &\rightarrow \text{Const} \mid \text{Var} \end{aligned}$$

Note that `<` and `<=` together with conjunction (which can be achieved by nested `if` statements) and disjunction (which can be achieved by multiple `if` statements in parallel), is enough to allow all of the standard comparisons to be expressed:

$$\begin{aligned} x < y &\iff x < y \\ x > y &\iff y < x \\ x \leq y &\iff x \leq y \\ x \geq y &\iff y \leq x \\ x = y &\iff x \leq y \wedge y \leq x \\ x \neq y &\iff x < y \vee y < x \end{aligned}$$

### 4.1.2 Example Program

Recall the example program from Chapter 1:

```
y = 1;
while (x > 0) {
   y = y * 2;
   x = x - 1;
}
```

A parse tree corresponding to this program (with intermediate non-terminals omitted when unambiguous) is given in Figure 4.1.

### 4.1.3 Semantics

The semantics of While programs are the same as C programs with the same syntax, except:

1. Execution starts at the first command of the program, and ends when the last command has been executed. For simplicity, there is currently no other way to halt execution, but we acknowledge that a `stop` command may be needed for some programs, and may be added in the future.
2. All variables exist in the same global scope throughout the execution of the program.
3. If a time limit is defined in the language bias (in the current system, one must be defined) and the number of instructions executed by the program exceeds that limit, the program will be considered to have failed to terminate, and such programs will not be considered as solutions.

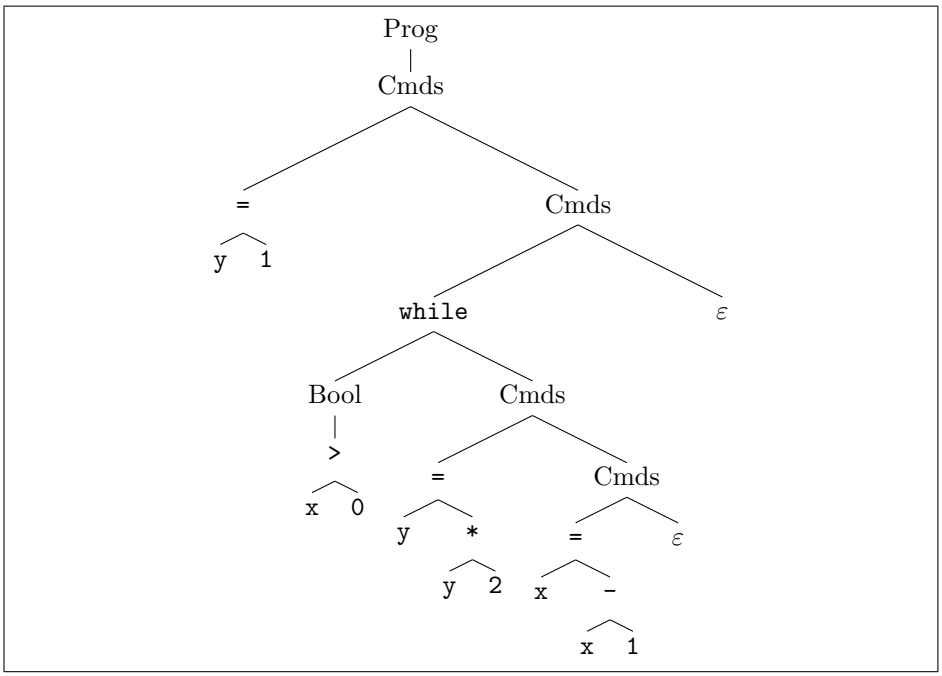


**Figure 4.1:** The parse tree of a While program.

4. A program attempting to perform arithmetic using a variable to which no value has been assigned will formally fail to terminate, therefore excluding such programs from being generated.
5. Arithmetic results exceeding the range of integers allowed by the language bias cause the program to formally fail to terminate.
6. Division by zero, and taking the remainder of division by zero, cause the program to formally fail to terminate.

Input and output is achieved by setting the initial values of certain *input variables* before the program runs, and reading the values of certain *output variables* after it has terminated, with the mechanics of how this is done depending on the context.

## 4.2 Program Encoding in ASP

In choosing an encoding of While programs as ASP facts, it is necessary to consider that we eventually want to search regions in the space of all possible programs, and that the program representation in ASP can have a significant effect on the efficiency of this search.

Acknowledging the fact that Answer Set Programs need to be fully grounded before evaluation, and to reduce the size of this grounding, it was chosen to use a "flat" representation of programs as a fixed-length sequence of discrete instructions. This is in contrast with the hierarchical representation of abstract syntax trees (ASTs) used elsewhere to reason about While programs, which due to their dynamic shape generated a large number of possible ground facts.

The chosen "flat" representation comes with the drawback that it it possible to represent nonsensical programs that don't correspond to any AST, but these configurations can be eliminated without excessive cost by the introduction of integrity constraints to enforce a hierarchical structure. See Chapter 5 for more details of how this is implemented.

Programs are represented as sequences of numbered *lines*, where each line contains an *instruction*. The fact that the instruction `Instr` occurs on line `L`, where `L` ranges from 1 to some maximum line number, is represented by:

```
line_instr(L, Instr).
```

The instruction `Instr` may be any of the following terms:

| | |
|---|---|
| `set(Var, Expr)` | The value of expression `Expr` is assigned to variable `Var`. |
| `if(Bool, Length)` | If the boolean guard `Bool` evaluates as true, the next `Length` instructions are executed; otherwise, they are skipped. |
| `while(Bool, Length)` | The next `Length` instructions are executed as many times as the guard `Bool` remains true. |
| `end_while` | This special instruction marks the end of `while` loop, and must be present immediately after the instructions encompassed by each loop. |

This representation bears a passing resemblance to the discrete, low-level instructions used in assembly languages (or machine bytecode), except that each instruction represents a higher-level program element, and so a greater amount structure can be enforced on their arrangement.

As the reader may have noted, the `end_while` is not strictly necessary, as its position could be inferred from the corresponding `while` instruction's `Length`; but it is included to simplify the implementation of a virtual machine for running these programs: when encountering an `end_while`, execution returns to the corresponding `while` to possibly repeat the loop, which is a discrete operation in itself. This is not necessary for `if` statements, so there is no `end_if` instruction.

Expressions `Expr`, which may occur on the right-hand side of `set` instructions and inside boolean guards, take the following forms, where `Expr1` and `Expr2` are also expressions:

| | |
|---|---|
| `var(Var)` | The value of a variable `Var` |
| `con(Con)` | The constant value `Con` |
| `add(Expr1, Expr2)` | The sum `Expr1 + Expr2` |
| `sub(Expr1, Expr2)` | The difference `Expr1 - Expr2` |
| `mul(Expr1, Expr2)` | The product `Expr1 * Expr2` |
| `div(Expr1, Expr2)` | The quotient `Expr1 / Expr2` |
| `mod(Expr1, Expr2)` | The remainder `Expr1 % Expr2` |

Boolean guards `Bool`, which control the execution of `if` and `while` instructions, take the following forms, where `Expr1` and `Expr2` are expressions:

| | | |
|---|---|---|
| `lt(Expr1, Expr2)` | less than: | `Expr1 < Expr2` |
| `le(Expr1, Expr2)` | less than or equal: | `Expr1 <= Expr2` |

Below is an example of the program from Chapter 1, which computes powers of two, represented in this encoding, with comments on the right giving each represented instruction in its more familiar form:

```
line_instr(1, set(y, con(1))).                % y = 1;
line_instr(2, while(lt(con(0), var(x)), 2)).  % while (0 < x) {
line_instr(3, set(y, mul(var(y), con(2)))).   %     y = y * 2;
line_instr(4, set(x, sub(var(x), con(1)))).   %     x = x - 1;
line_instr(5, end_while).                     % }
```

## 4.3 Running Programs in ASP

We have implemented in ASP an interpreter for the While language, `run.lp`, which when combined with a set of facts giving a program such as the above, and further facts specifying the initial values of variables, computes an answer set containing facts giving the final values of variables after running the program. The constants `time_max`, `int_min` and `int_max` must also be specified by the user, to effect suitable execution limits and to allow the program's grounding to be finite.

`run.lp` allows multiple execution traces to be computed at once, using the predicate `in/3`, which takes the following form:

```
in(RunID, Variable, InitialValue).
```

For each execution trace a unique `RunID` is assigned, and one or more instances of `in/3` are defined, one for each input variable.

The answer set contains facts of the following form:

```
run_var_out(RunID, Variable, FinalValue)
run_var_out(RunID, Variable, unset)
run_does_not_halt(RunID)
```

The first signifies that `RunID` terminated successfully with the given final value for a variable; the second signifies that the run terminated successfully, but without assigning a value for a variable; and the last signifies that the run did not successfully terminate, which could be because the number of instructions executed exceeded `time_max` (which would happen if the program contained an infinite loop, for example), or because the value of a variable exceeded the range defined by `int_min` and `int_max`, or because of another illegal operation such as division by zero.

### Implementation

In order to achieve this, `run.lp` computes for every run `R`, for every time `T` with $0 \leq$ `T` $<$ `time_max`, the line `L` of the program currently being executed, and the value of every variable. A particular run is considered to have terminated successfully if an `L` is reached for which there is no `line_instr`, i.e. the end of the program has been reached.

Also needed are the values of arithmetic expressions `E` and boolean expressions `B` for certain runs `R` and at certain times `T`. The fact `run_expr(R, T, E, C)` means that the expression represented by the atom `E` evaluates to the integer `C`. The fact `run_bool(R, T, B)` means that the boolean expression `B` evaluates as true, and the absence of this fact means that it evaluates as false. The rules implementing these predicates are lengthy but straightforward, and so their implementation is omitted from this chapter for brevity.

Also defined are the domain predicates `run/1`, `int/1` and `var/1` listing the run IDs, the legal integer values and the existing variables, which are derived straightforwardly from the aforementioned constants and from `line_instr/2`.

The line number `L` for run `R` at time `T` is represented by `run_line(R, T, L)`, and computed using the following rules:

```
% Identify lines where execution may jump to a remote point.
line_jump(L) :- line_instr(L, if(_,_)).
line_jump(L) :- line_instr(L, while(_,_)).
line_jump(L) :- line_instr(L, end_while).

% For efficiency, project out if/while parameters.
line_instr_if_guard(L, G)     :- line_instr(L, if(G, _)).
line_instr_if_length(L, B)    :- line_instr(L, if(_, B)).
line_instr_while_guard(L, G)  :- line_instr(L, while(G, _)).
line_instr_while_length(L, B) :- line_instr(L, while(_, B)).

% Start at line 1, and by default move to the next line at each step.
run_line(R,0,1) :-
    run(R).
run_line(R,T+1,L+1) :-
    line(L), run(R,T), run_line(R,T,L),
    not line_jump(L), not run_fail(R,T).

% Run if statements.
run_line(R,T+1,L+1) :-
    run(R,T), run_line(R,T,L), line_instr_if_guard(L,B),
    run_bool(R,T,B).
run_line(R,T+1,L+BodyLength+1) :-
    run(R,T), run_line(R,T,L), line_instr(L,if(B,BodyLength)),
    not run_bool(R,T,B).

% Run while loops.
run_line(R,T+1,L+1) :-
    % Boolean guard is true upon (re-)entering loop.
    run(R,T), run_line(R,T,L), line_instr_while_guard(L,B),
    run_bool(R,T,B).
run_line(R,T+1,L+BodyLength+2) :-
    % Boolean guard is false upon (re-)entering loop.
    run(R,T), run_line(R,T,L), line_instr(L,while(B,BodyLength)),
    not run_bool(R,T,B).
run_line(R,T+1,HeadLine) :-
```

```
    % Re-enter loop after reaching end_while.
    run(R,T), run_line(R,T,L), line_instr(L,end_while),
    line_instr_while_length(HeadLine, BodyLength),
    L == HeadLine+BodyLength+1.
```

The value `C` of the variable `X` in run `R` at time `T` is represented by `run_var(R, T, X, C)`, and is implemented by the following rules:

```
% Initialise variables to their input value, if any.
run_var(R,0,X,C) :- in(R,X,C), int(C).

% Identify lines where a particular variable's value may change.
line_set(L,X) :- line_instr(L, set(X,_)).

% On a set statement, update variables; otherwise, leave them constant.
run_var(R,T+1,X,C) :-
    run_line(R,T,L), not line_set(L,X),
    run_var(R,T,X,C), int(C).
run_var(R,T+1,X,C) :-
    run_line(R,T,L), line_set(L,X), line_instr(L,set(X,E)),
    run_expr(R,T,E,C), int(C).

% Identify expressions which evaluate to a legal result.
run_expr_not_fail(R,T,E) :- run_expr(R,T,E,_).

% Identify the time at which a run fails due to illegal arithmetic.
run_fail(R,T) :-
    run_line(R,T,L), line_set(L,X), line_instr(L,set(X,E)),
    not run_expr_not_fail(R,T,E).
```

Finally, in order to make use of the computed results and to define `run_var_out` and `run_does_not_halt` mentioned above, some further computation is needed, and is implemented by the following rules:

```
% run(R,T) if test case R runs to at least time T.
run(R,T) :- run_line(R,T,L), T<time_max, line(L).

% run_halt(R,T) if test case R halts normally at time T.
run_halt(R,T) :- run_line(R,T,L), T<time_max, not line(L).

run_does_halt(R) :- run_halt(R,_).
run_does_not_halt(R) :- run(R), not run_does_halt(R).

% run_var_out(R,X,C) if the final value of X in run R is C.
run_var_out(R,X,C)     :- run_var(R,T,X,C), not run(R,T).
run_var_out(R,X,unset) :- run(R), var(X), not run_var_out_set(R,X).
run_var_out_set(R,X)   :- run_var_out(R,X,C), C != unset.
```

## 4.4 Example Programs

We now introduce some examples of programs that can be run using `run.lp` and their sample inputs and outputs. We will return to some of these same programs in later chapters to show how they can be automatically generated.

### Powers of Two

When we run `run.lp` with the constants `time_max=99`, `int_min=0`, `int_max=32`, with the facts from the example at the end of Section 4.2:

```
line_instr(1, set(y, con(1))).                % y = 1;
line_instr(2, while(lt(con(0), var(x)), 2)).  % while (0 < x) {
line_instr(3, set(y, mul(var(y), con(2)))).   %     y = y * 2;
line_instr(4, set(x, sub(var(x), con(1)))).   %     x = x - 1;
line_instr(5, end_while).                     % }
```

And with the following additional facts:

```
in(run0,x,0).
in(run1,x,1).
in(run2,x,2).
in(run3,x,3).
in(run4,x,4).
in(run5,x,5).
```

Then the resulting answer set computed by Clingo 3 contains the following instances of `run_var_out/3`:

```
run_var_out(run0,x,0)  run_var_out(run0,y,1)
run_var_out(run1,x,0)  run_var_out(run1,y,2)
run_var_out(run2,x,0)  run_var_out(run2,y,4)
run_var_out(run3,x,0)  run_var_out(run3,y,8)
run_var_out(run4,x,0)  run_var_out(run4,y,16)
run_var_out(run5,x,0)  run_var_out(run5,y,32)
```

Which as we would expect gives the correct result, with $y = 2^{x_0}$, where $y$ is the final value of `y` and $x_0$ is the initial value of `x`.

### Triangular Numbers

The $n$th triangular number is defined as $T_n = \sum_{i=0}^{n} i$. There is also a closed formula, $T_n = \frac{n(n+1)}{2}$. Therefore, the following two programs are functionally equivalent:

```
line_instr(1, set(t, add(var(n), con(1)))).   % t = n + 1;
line_instr(2, set(t, mul(var(t), var(n)))).   % t = t * n;
line_instr(3, set(t, div(var(t), con(2)))).   % t = t / 2;
```

```
line_instr(1, set(t, con(0))).                % t = 0;
line_instr(2, while(lt(con(0), var(n)), 2)).  % while (n > 0) {
line_instr(3, set(t, add(var(t), var(n)))).   %     t = t + n;
line_instr(4, set(n, sub(var(n), con(1)))).   %     n = n - 1;
line_instr(5, end_while).                     % }
```

Both of the above programs can be run with the following parameters:

```
#const time_max=50.
#const int_min=0.
#const int_max=50.

in(run0,n,0).
in(run1,n,1).
in(run2,n,2).
in(run3,n,3).
in(run4,n,4).
in(run5,n,5).
```

To produce the following output, the familiar sequence of triangular numbers whose sequence of differences between successive terms is the natural numbers:

```
run_var_out(run0,t,0)
run_var_out(run1,t,1)
run_var_out(run2,t,3)
run_var_out(run3,t,6)
run_var_out(run4,t,10)
run_var_out(run5,t,15)
```

**Fibonacci Sequence**

The Fibonacci Sequence $(F_n)_{n=1}^{\infty}$, given by $F_1 = F_2 = 1$, $F_{n+2} = F_n + F_{n+1}$, can be computed by the following program, which takes as input `n`, and produces `f` $= F_{\texttt{n}}$, the `n`th term in the sequence:

```
line_instr(1, set(e, con(0))).                % e = 0;
line_instr(1, set(f, con(1))).                % f = 1;
line_instr(2, while(lt(con(1), var(n)), 3)).  % while (n > 1) {
line_instr(3, set(f, add(var(e), var(f)))).   %    f = e + f;
line_instr(4, set(e, sub(var(f), var(e)))).   %    e = f - e;
line_instr(5, set(n, sub(var(n), con(1)))).   %    n = n - 1;
line_instr(6, end_while).                     % }
```

We can run this to compute the first 6 terms of the sequence using the following parameters:

```
#const time_max=30.
#const int_min=0.
#const int_max=10.

in(run1,n,1).
in(run2,n,2).
in(run3,n,3).
in(run4,n,4).
in(run5,n,5).
in(run6,n,6).
```

Which results in an answer set containing the following results:

```
run_var_out(run1,n,1) run_var_out(run1,e,0) run_var_out(run1,f,1)
run_var_out(run2,n,1) run_var_out(run2,e,1) run_var_out(run2,f,1)
run_var_out(run3,n,1) run_var_out(run3,e,1) run_var_out(run3,f,2)
run_var_out(run4,n,1) run_var_out(run4,e,2) run_var_out(run4,f,3)
run_var_out(run5,n,1) run_var_out(run5,e,3) run_var_out(run5,f,5)
run_var_out(run6,n,1) run_var_out(run6,e,5) run_var_out(run6,f,8)
```

In the third column, we can see the final values of `f`, which indeed form the familiar Fibonacci sequence. The second column, with the values of `e`, forms the same sequence but shifted forward by one space, which offers an insight into how the program works.

### Median of Three Numbers

Given three real numbers $a$, $b$ and $c$, their median is the unique real number $m \in \{a, b, c\}$ such that $\min(a, b, c) \leq m \leq \max(a, b, c)$. An example of a program that finds the median of three integers is the following, which puts `a`, `b` and `c` in ascending order using `m` as a swap variable, then sets `m` to `b`, the middle value:

```
line_instr( 1, if(lt(var(c), var(a)), 3)).  % if (a > c) {
line_instr( 2, set(m, var(a))).             %     m = a;
line_instr( 3, set(a, var(c))).             %     a = c;
line_instr( 4, set(c, var(m))).             %     c = m; }
line_instr( 5, if(lt(var(b), var(a)), 3)).  % if (a > b) {
line_instr( 6, set(m, var(a))).             %     m = a;
line_instr( 7, set(a, var(b))).             %     a = b;
line_instr( 8, set(b, var(m))).             %     b = m; }
line_instr( 9, if(lt(var(c), var(b)), 3)).  % if (b > c) {
line_instr(10, set(m, var(b))).             %     m = b;
line_instr(11, set(b, var(c))).             %     b = c;
line_instr(12, set(c, var(m))).             %     c = m; }
line_instr(13, set(m, var(b))).             % m = b;
```

Configured with the following parameters:

```
#const time_max=10.
#const int_min=0.
#const int_max=10.

in(r123,a,11). in(r123,b,12). in(r123,c,13).
in(r132,a,21). in(r132,b,23). in(r132,c,22).
in(r312,a,33). in(r312,b,31). in(r312,c,32).
in(r321,a,43). in(r321,b,42). in(r321,c,41).
in(r231,a,52). in(r231,b,53). in(r231,c,51).
in(r213,a,62). in(r213,b,61). in(r213,c,63).
```

Produces the following output:

```
run_var_out(r123,m,12)
run_var_out(r132,m,22)
run_var_out(r312,m,32)
run_var_out(r321,m,42)
run_var_out(r231,m,52)
run_var_out(r213,m,62)
```

### $n$th Prime Number

A positive integer $p$ is prime if $p > 1$ and $p$ has no proper divisors, i.e. there are no two integers $k, j$ such that $1 < k < p$ and $p = kj$. Euclid's Theorem, a result in elementary number theory, states that there are infinitely many prime numbers; so we can define a sequence $(p_n)_{n=1}^{\infty} = (2, 3, 5, 7, ...)$, where $p_n$ is the $n$th prime number. There are known a number of efficient algorithms to compute $p_n$ given $n$, but the following program implements a relatively simple one:

```
line_instr( 1, set(p, con(2))).                % p = 2;
line_instr( 2, if(lt(con(1), var(n)), 2)).     % if (n > 1) {
line_instr( 3, set(p, con(3))).                %   p = 3;
line_instr( 4, set(n, sub(var(n), con(1)))).   %   n = n - 1; }
line_instr( 5, while(lt(con(1), var(n)), 12)). % while (n > 1):
line_instr( 6, set(n, sub(var(n), con(1)))).   %   n = n - 1;
line_instr( 7, set(r, con(0))).                %   r = 0;
line_instr( 8, while(le(var(r), con(0)), 8)).  %   while (r <= 0):
line_instr( 9, set(p, add(var(p), con(2)))).   %     p = p + 2;
line_instr(10, set(d, con(3))).                %     d = 3;
line_instr(11, while(lt(var(d), var(p)), 4)).  %     while (d < p) {
line_instr(12, set(r, mod(var(p), var(d)))).   %       r = p % d;
line_instr(13, set(d, add(var(d), con(2)))).   %       d = d + 2;
line_instr(14, if(le(var(r), con(0)), 1)).     %       if (r <= 0) {
line_instr(15, set(d, var(p))).                %         d = p; }
line_instr(16, end_while).                     %     }
line_instr(17, end_while).                     %   }
line_instr(18, end_while).                     % }
```

Running this program with the following parameters:

```
#const time_max=100.
#const int_min=0.
#const int_max=15.

in(run1,n,1).
in(run2,n,2).
in(run3,n,3).
in(run4,n,4).
in(run5,n,5).
in(run6,n,6).
```

We obtain an answer set containing the following results:

```
run_var_out(run1,p,2)
run_var_out(run2,p,3)
run_var_out(run3,p,5)
run_var_out(run4,p,7)
run_var_out(run5,p,11)
run_does_not_halt(run6)
```

The first 5 runs successfully computed the value we would expect, but `run6` failed to halt. This is because the number of time-steps needed to count up to the 6th prime number exceeds our value of `time_max=100`. It is possible to raise this value and obtain a value of $\mathtt{p} = 13$, but at this point the grounding of the ASP becomes large enough that extending it to a search of many similar programs of this complexity is computationally infeasible when running on current desktop machines.

This indicates that a divide-and-conquer approach will be needed in order to address more complicated programs. The above program can be seen to consist of three main subprograms: the innermost loop, which checks whether $\mathtt{p} \geq 3$ is a prime number; the loop containing that, which increments a prime number $\mathtt{p} \geq 3$ to the next largest prime number; and the outermost loop, which increments $\mathtt{p} \geq 3$ to the $(n-1)$th next largest prime number. It is possible to reason separately about the correctness of these subprograms, so in the same way it may be possible to synthesise them separately.

# Chapter 5

# Synthesis of Programs from User Examples

The kernel of inductive program synthesis is to take a list of input/output examples and find a program satisfying them. This chapter will detail the subset of our work implementing this scheme. In later chapters we will describe how we built upon the idea to effect improvements.

For now, it consists of two components:

- An Answer Set Program, `learn.lp`, which contains a modified version of `run.lp` from Section 4.3, in addition to aggregrates and integrity constraints which invert the computation done by `run.lp` so that it takes program results as *input*, and produces the program as *output*.
- A Haskell program `IterativeLearn` which reads a *task specification* and sets up parameters to `learn.lp`, running it multiple times if necessary with different parameters to find a program meeting the task specification, and prints it in a human-readable form.

## 5.1 Task Specification Language

The user specifies a program synthesis task by writing a configuration file containing the relevant parameters. The format chosen for this file is an Answer Set Program whose answer set contains facts giving the parameters.

As long as their names do not clash with ones used in `learn.asp`, additional predicates may be specified in this file, and referred to from the configuration parameters.

The following categories of parameters may be given:

### 5.1.1 Input/Output Examples

```
input_variable(V1; ...; Vn).
```

The n variables {V1, ..., Vm} are declared as input variables, meaning that they are initialised with values to parameterise an individual execution of the program.

```
output_variable(V1; ...; Vn).
```

The n variables {V1, ..., Vm} are declared as output variables, meaning that their values after the program terminates are examined, and possibly compared with examples to determine program correctness.

```
 in(ExampleID, InputVariable,  InitialValue).
out(ExampleID, OutputVariable, ExpectedFinalValue).
```

These parameters allow the user to list input/output examples that the program is required to satisfy. Any number of examples may be given, and each set of examples should have a unique ExampleID, with which each InputVariable and OutputVariable is associated with an InitialValue or ExpectedFinalValue.

### 5.1.2 Language Bias

```
constant(C1; ...; Cn).           % Optional; default: no constants.
```

The n numerical constants {C1, ..., Cn} given here may occur as literals in the program. No other constants may occur.

```
extra_variable(V1; ...; Vn).     % Optional; default: no extra variables.
```

The n variables {V1, ..., Vn} given here may occur in the program in addition to those given by input_variable1 and output_variable1. No other variables may occur.

```
read_only_variable(V1; ...; Vn). % Optional; default: all variables writable.
```

None of the n variables {V1, ..., Vn} given here may be modified by the program. Of course, this only makes sense for input variables, as a read-only variable could only otherwise hold a value by receiving an input value. This can be used to speed up the search for a program by eliminating spurious variable assignments.

```
disallow_feature(if).           % No if statements.
disallow_feature(while).        % No while loops.

disallow_feature(add).          % No addition.
disallow_feature(sub).          % No subtraction.
disallow_feature(mul).          % No multiplication.
disallow_feature(div).          % No integer division.
disallow_feature(mod).          % No remainder.
disallow_feature(arithmetic).   % None of the above operations.
```

The given syntactic element is prevented from occurring in a generated program, unless it occurs in user-specified code in the program template.

### 5.1.3 Program Template

```
preset_line_instr(Line, Instr).
preset_line_instr(line_max-N, Instr).
```

In the first form, an instruction at a certain line number is fixed, using the same format as `line_instr` from Section 4.3. In the second form, the special constant `line_max`, which is instantiated to the number of lines in the program the synthesiser is currently trying to generate, is used to specify an instruction `N` lines from the end of the program.

### 5.1.4 Execution Limits

```
int_range(Imin, Imax).  % Required.
```

The value of no variable may be less than `Imax` or greater than `Imin`.

```
time_limit(Tmax).       % Required.
```

No more than `Tmax` instructions may be executed in any single execution.

### 5.1.5 Search Hints

```
line_limit_min(Lmin).   % Optional; default: 0.
line_limit_max(Lmax).   % Optional; default: unbounded.
line_limit_step(Lstep). % Optional; default: 1.
```

Controls the incremental search for a program with increasing limits on the number of lines. Initially, a program with `Lmin` lines is sought (for the default, `Lmin=0`, this is the empty program). If no such program can be found, the limit is increased by `Lstep` and the search repeated, until the limit exceeds `Lmax`.

## 5.2 Implementation

### 5.2.1 `learn.lp`

In `learn.lp`, The Answer Set Program `run.lp` from Section 4.3 is modified to generate and then search the domain of programs fitting the given language bias and resource restrictions.

Given input/output examples expressed by several of the following facts:

```
 in(ExampleID, InputVariable, Value).
out(ExampleID, OutputVariable, ExpectedValue).
```

The part of the answer set we are now interested in is the generated program, given as one instruction per line:

```
line_instr(LineNumber, Instruction).
```

This is in contrast to `run.lp` which takes `line_instr/2` as given, and generates an answer set containing `run_var_out/3` facts giving its output.

The following aggregate expresses that there must be exactly one instruction per line, but gives the solver the choice of any valid instruction at a given line, except for preset lines. The constant `line_max` is exactly equal to the number of lines in the program we are trying to generate.

```
1{ line_instr(L, I) : valid_line_instr(L, I) }1 :- line(L).
line_instr(L, I) :- preset_line_instr(L, I).
line(1..line_max).
```

Then, in order to direct the search to the program we want, the following integrity constraints are defined to assert that the program must terminate successfully with the correct output values:

```
:- run_does_not_halt(R).
:- run_var_out(R,V,Actual), out(R,V,Expected), Actual != Expected.
```

What constitutes a valid instruction depends on the line at which it is to occur, and is defined by the following rule. `write_var/1` and `allow/1` are the respective negations of `read_only_variable/1` and `disallow_feature/1` from the task specification with respect to suitable defaults. `expr/1` and `bool/1` define the valid arithmetic and boolean expressions.

```
% Setting a writable variable:
valid_line_instr(L, set(V, E)) :-
    line(L), write_var(V), expr(E).

% If statements (must end inside the program):
valid_line_instr(L, if(B, 1..M)) :-
    line(L), allow(if), bool(B), M=line_max-L.

% While loops (must end inside the program):
valid_line_instr(L, while(B, 1..M)) :-
    line(L), allow(while), bool(B), M=line_max-1.

% While loop terminators.
valid_line_instr(L, end_while) :-
    line(L), allow(while).

% Allow any pre-set line unconditionally:
valid_line_instr(L, I) :-
    preset_line_instr(L, I).
```

In addition, we need to contrain away nonsensical combinations of instructions, which is achieved by the following integrity constraints:

```
% We need to project out the Length parameter from line_instr/2:
line_instr_if_length(L, Len)    :- line_instr(L,    if(_, Len)).
line_instr_while_length(L, Len) :- line_instr(L, while(_, Len)).

% Forbid if/while blocks that end outside of a block they start within:
:- line_instr_if_length(L,M),
   block(PL,PM), PL<L, L<=PL+PM, L+M>PL+PM.
:- line_instr_while_length(L,M),
   block(PL,PM), PL<L, L<=PL+PM, L+M+1>PL+PM.
% ...where a "block" is an if statement or a while loop:
block(L, M) :- line_instr(L,    if(_, M)).
block(L, M) :- line_instr(L, while(_, M))
```

```
% Require end_while statements to occur at the correct positions:
:- line_instr(L,end_while), not end_while_block(L).
:- end_while_block(L), not line_instr(L,end_while).
% ... where the "correct positions" are given by:
end_while_block(L) :- line_instr_while_length(LH,B), L=LH+B+1.
```

Finally, the predicates `expr/1` and `bool/1` need to be defined, to generate the domain of possible expressions. The grounding size of `learn.lp` is sensitive to how this is done, so care was taken to eliminate redundant expressions wherever possible.

The implementation is lengthy, so we give an example for `add` expressions and refer to the reader to the full text of `learn.lp` for details of other operations:

```
% x+c makes sense if c is positive (but x+0 does not),
% but since addition is commutative, we exclude c+x:
expr(add(var(X), con(C))) :-
    allow(add), var(X), con(C), C > 0.

% x+(-c) only makes sense if x-c is not possible:
expr(add(var(X), con(C))) :-
    allow(add), var(X), con(C), C < 0, not allow(sub).
expr(add(var(X), con(C))) :-
    allow(add), var(X), con(C), C < 0, not con(-C).

% By commutativity, allow exactly one of x+y or y+x:
expr(add(var(X), var(Y))) :-
    allow(add), var(X), var(Y), X < Y.

% x+x only makes sense if 2*x is not possible:
expr(add(var(X), var(X))) :-
    allow(add), var(X), not allow(mul).
expr(add(var(X), var(X))) :-
    allow(add), var(X), not con(2).
```

### 5.2.2 `IterativeLearn`

The purpose of the program `IterativeLearn` is simply to execute `learn.lp` with the correct parameters and to provide a somewhat more abstracted user interface.

Subject to the parameters `line_limit_min`, `line_limit_step` and `line_limit_max` (Section 5.1.5), it incrementally searches for a program with the smallest possible number of lines. In fact, it would have been possible to implement the same functionality using ASP *optimisation statements* in `learn.lp` or using `iclingo`, an iterative version of Clingo from the Potassco[11] tools. However, in an extended version of the system where `IterativeLearn`'s mediation in the search process is more involved, neither of these alternatives would work.

When a program is successfully generated, `IterativeLearn` converts the set of facts representing the program into a more human-readable representation of the program's syntax, presents this to the user, and exits.

## 5.3 Synthesising Example Programs

We now demonstrate the system as applied to some example synthesis tasks.

### Powers of Two

This task specification includes the first 6 powers of two as input/output examples, and enforces the reasonable constraint that the only numerical constants to occur in the code are `1` and `2`:

```
% pow2_ex.lp
int_range(0, 32).
time_limit(10).
constant(1; 2).

input_variable(x).
output_variable(y).

in(ex0, x, 0). out(ex0, y, 1).
in(ex1, x, 1). out(ex1, y, 2).
in(ex2, x, 2). out(ex2, y, 4).
in(ex3, x, 3). out(ex3, y, 8).
in(ex4, x, 4). out(ex4, y, 16).
in(ex5, x, 5). out(ex5, y, 32).
```

When run with `IterativeLearn`, the following output is produced:

```
$ ./IterativeLearn.hs ../examples/iterative/pow2_ex.lp
Searching for a program with 0 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 1 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 2 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 3 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 4 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 5 lines satisfying 5 example(s)...
Found the following program:
   1. y = 1
   2. while (1 <= x):
   3.     y = 2 * y
   4.     x = x - 1
   5. end_while
```

### Triangular Numbers

A specification by example of the program computing the triangular numbers $\mathtt{T} = \sum_{i=0}^{\mathtt{n}} i$ is given below:

```
% triangle_ex.lp
time_limit(30).
int_range(0, 30).
constant(0; 1; 2).

input_variable(n).
output_variable(t).

in(r0,n,0). out(r0,t,0).
in(r1,n,1). out(r1,t,1).
in(r2,n,2). out(r2,t,3).
in(r3,n,3). out(r3,t,6).
in(r4,n,4). out(r4,t,10).
```

```
$ ./IterativeLearn.hs ../examples/iterative/triangle_ex.lp
Searching for a program with 0 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 1 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 2 lines satisfying 5 example(s)...
No such program found.
Searching for a program with 3 lines satisfying 5 example(s)...
Found the following program:
   1. t = n * n
   2. n = n + t
   3. t = n / 2
```

This gives a program equivalent to the 3-line example in Section 4.4, except it computes the polynomial formula by adding together terms instead of by multiplying together factors.

If we like, we can add `read_only_variable(n)` to the specification, preventing `n` from being used as a temporary variable in this way, and we obtain instead:

```
Found the following program:
   1. t = n + 1
   2. t = n * t
   3. t = t / 2
```

Which is now exactly the same program that we manually wrote.

Alterantively, we can try to generate a program that computes the triangular numbers by adding in a loop, by using the following parameters with the same examples as above:

```
time_limit(30).
int_range(0, 10).
constant(0; 1).

input_variable(n).
output_variable(t).

disallow(if; mul; div; mod).
```

And we obtain:

```
Found the following program:
   1. t = n
   2. while (1 <= n):
   3.     n = n - 1
   4.     t = n + t
   5. end_while
```

Which is more or less the same as the handwritten implementation, albeit using an unusual order of summation.

**Median of Three Numbers**

The task of giving the median value of three integers is expressed by example in the following task specification. We disable arithmetic operations and while loops because we know the median can be computed in constant time and without arithmetic.

We make the inputs read-only, and provide two general-purpose variables for writing in addition to the output variable `m`. We also know that there's no point considering short programs, so we place a conservative minimum of 5 on the program's length.

```
time_limit(12).
int_range(0, 7).
line_limit_min(5).

input_variable(a; b; c).
read_only_variable(a; b; c).
output_variable(m).
extra_variable(x; y).

disallow_feature(while; arithmetic).

% Examples:
ex(r00, 0,0,0, 0).
ex(r10, 7,1,1, 1).
ex(r20, 1,7,1, 1).
ex(r30, 1,1,7, 1).
ex(r40, 1,7,7, 7).
ex(r50, 7,1,7, 7).
ex(r60, 7,7,1, 7).
ex(r70, 1,2,3, 2).
ex(r80, 1,3,2, 2).
ex(r90, 2,1,3, 2).
ex(rA0, 3,2,1, 2).
ex(rB0, 2,3,1, 2).
ex(rC0, 3,1,2, 2).

% This predicate allows examples to be written compactly.
 in(R,a,A) :- ex(R,A,_,_,_).
 in(R,b,B) :- ex(R,_,B,_,_).
 in(R,c,C) :- ex(R,_,_,C,_).
out(R,m,M) :- ex(R,_,_,_,M).
```

However, when we try to run this specification, the complexity of the search causes it to become unreasonably slow, and, impatient and having a vague idea of how the program will be structured, we decide to specify the first three lines of the program to speed up the search:

```
% Set x = max(a, b).
preset_line_instr(1, set(x, var(a))).
preset_line_instr(2, if(lt(var(x), var(b)), 1)).
preset_line_instr(3, set(x, var(b))).
```

With this addition to the task specification, the following result can now be computed in reasonable time:

```
Searching for a program with 5 lines satisfying 13 example(s)...
No such program found.
Searching for a program with 6 lines satisfying 13 example(s)...
No such program found.
Searching for a program with 7 lines satisfying 13 example(s)...
No such program found.
Searching for a program with 8 lines satisfying 13 example(s)...
No such program found.
Searching for a program with 9 lines satisfying 13 example(s)...
No such program found.
Searching for a program with 10 lines satisfying 13 example(s)...
Found the following program:
   1. x = a
   2. if (x < b):
   3.     x = b
   4. m = a
   5. if (c <= x):
   6.     if (b < a):
   7.         m = b
   8.     x = c
   9. if (m < x):
  10.  m = x
```

Although it's not a particularly easy program to read, we can, analysing its cases, see that it computes $m = \min(\max(a,b), \max(b,c), \max(c,a))$, which, being $\leq \max(a,b,c)$ and $\geq \min(a,b,c)$, is the median of $a$, $b$ and $c$, so it meets its intended specification.

# Chapter 6

# Automatic Generation of Examples

The system present so far relies on the user to give examples of the program's behaviour, but it's quite possible to give a bad set of examples that doesn't fully represent the program the user has in mind, or to give more examples than necessary and slow down the generation process.

We now extend the system with the ability to take a functional specification that expresses exactly the program we want, and uses this to generate examples until a set of examples is accumulated yielding a program that meets the specification.

## 6.1 Extensions to the Task Specification Language

The specification language from Chapter 5 is extended with the following configuration entries:

### 6.1.1 Functional Specification

```
precondition(PreconditionString).   % Optional; default: true.
postcondition(PostconditionString). % Optional; default: true.
```

If a precondition and/or postcondition are given, the task is to synthesise a program whose output values satisfy the postcondition whenever its input values satisfy the precondition, in addition to any constraints given by input/output examples.

Preconditions and postconditions are given as ASP `"strings"` containing propositional formulae written in the same syntax as the body of an ASP rule. Preconditions may contain no ASP variables other than those corresponding to the initial values of input variables, which take the form `In_x` for an input variable named `x`. Postconditions may contain the same ASP variables, in addition to

those corresponding to the final values of output variables, which take the form `Out_y` for an output variable named `y`.

## 6.2 Implementation

A new component is introduced: a dynamically generated answer set program based on `counterexample.lp`, which uses `run.lp` and takes an existing program, a precondition and a postcondition, and attempts to find an input satisfying the precondition but whose resulting output does not satisfy the postcondition. If this happens, the existing program is incorrect, and we also calculate what the output should have been for this input.

`IterativeLearn` now alternates between generating a program from the examples recorded so far, and searching for a new example to falsify the current program and extend its list of examples. Once no more examples can be found, the program is considered correct and offered as a solution. This results in each example being relevant to the process, and the number of examples being no greater than necessary. It also gives stronger confidence that the resulting program is correct, subject to the configured `int_range` of values being considered.

## 6.3 Synthesising Example Programs

**Powers of Two**

Returning to the familiar example of computing powers of two, a specification is given below that uses no examples and just a precondition and postcondition expressing the problem:

```
int_range(0, 32).
time_limit(10).
constant(1; 2).

input_variable(x).
output_variable(y).

precondition("In_x >= 0").
postcondition("Out_y == 2 ** In_x").
```

Running this with the modified `IterativeLearn`, the following output is produced:

```
$ ./IterativeLearn.hs ../examples/iterative/pow2_con.lp
Searching for a program with 0 lines satisfying 0 example(s)...
Found the following program:
   (empty program)
Searching for a counter-example to falsify the postcondition...
Found the following counterexample:
   Input:    x = 5
   Expected: y = 32
   Output:   (none)
Searching for a program with 0 lines satisfying 1 example(s)...
```

```
No such program found.
Searching for a program with 1 lines satisfying 1 example(s)...
No such program found.
Searching for a program with 2 lines satisfying 1 example(s)...
No such program found.
Searching for a program with 3 lines satisfying 1 example(s)...
Found the following program:
   1. y = x * x
   2. y = y + 2
   3. y = x + y
Searching for a counter-example to falsify the postcondition...
Found the following counterexample:
   Input:    x = 4
   Expected: y = 16
   Output:   y = 22
Searching for a program with 3 lines satisfying 2 example(s)...
No such program found.
Searching for a program with 4 lines satisfying 2 example(s)...
No such program found.
Searching for a program with 5 lines satisfying 2 example(s)...
Found the following program:
   1. y = x - 2
   2. x = x + 1
   3. y = x * y
   4. y = y - 2
   5. y = 2 * y
Searching for a counter-example to falsify the postcondition...
Found the following counterexample:
   Input:    x = 3
   Expected: y = 8
   Output:   y = 4
Searching for a program with 5 lines satisfying 3 example(s)...
Found the following program:
   1. y = 2
   2. while (1 < x):
   3.     x = x - 1
   4.     y = 2 * y
   5. end_while
Searching for a counter-example to falsify the postcondition...
Found the following counterexample:
   Input:    x = 0
   Expected: y = 1
   Output:   y = 2
Searching for a program with 5 lines satisfying 4 example(s)...
Found the following program:
   1. y = 1
   2. while (1 <= x):
   3.     x = x - 1
   4.     y = 2 * y
   5. end_while
```

```
Searching for a counter-example to falsify the postcondition...
Success: the postcondition could not be falsified.
```

Rather than the 6 examples manually specified before, this approach found the program using 4 well-chosen examples. With too few examples, the program synthesiser just produced polynomial functions interpolating them; then with another example, an almost-correct program that failed at the sometimes overlooked case of `x` = 0; with that example added, the correct program was produced.

# Chapter 7

# Conclusions and Future Work

We have evaluated the applicability of Answer Set Programming to the task of inductively synthesising imperative programs, and the progress made in doing this indicates that the technique has some promise.

However, There are a lot of possible improvements or extensions to the system that have been considered, but not yet been thoroughly investigated. For example:

- Currently, programs can be specified only using a precondition and a postcondition; however, if it were allowed to annotate a program template with midconditions, the system could split the task into many parallel tasks involving learning subprograms. If successful, this should afford a tremendous performance improvement for large programs, and make it feasible to synthesise larger programs and over larger domains of values.
- Further the previous point, it should also be possible to annotate `while` loops in the program template with an *invariant* which should hold before and after each iteration, and a *variant* expression related to the loop's boolean guard which must strictly decrease on each iteration, and eventually falsify the guard. For a correct invariant and variant, this would allow a correct program to be generated faster. For an incorrect invariant or variant, the failure to generate the correct program might provide useful insight to the user about their problem.
- If it is possible to reason about subprograms in a separated manner, this could be further extended to formally introduce the notion of subroutines into the language definition. Practically, speaking, it's almost always necessary to divide a large imperative program into subroutines to avoid a repetitive and unreadable program, so this may be needed to make the tool applicable in practice. Allowing the user to define separate subroutines seems certainly within the realm of possiblity, and having the system invent its own subroutines could be even more powerful, but its feasibility is currently uncertain.

- The language of programs generated is currently quite restricted, and this limits the usefulness of the tool in practice. More data types and operations could be added to the existing language, or the applicability of these techniques to a different type of language, perhaps a functional programming language, could be investigated.
- The field of Inductive Logic Programming (ILP) involves the generation of logic programs, and in particular there has been work in inductively generating Answer Set Programs[14]. This project was initially envisioned to use ILP as part of the process of generating imperative programs, with the imperative program exracted by some further mechanism from a generated logic program. The project took a different direction due to the pure use of ASP appearing more immediately promising; nonetheless, it might be worthwhile to give further consideration to this notion, to see if it offers any improvements over the current system.
- The current system uses Clingo 3 for evaluating Answer Set Programs; however, there is a later version, Clingo 4, which uses an updated ASP language with additional features. It may be worth rewriting the current tools in Clingo 4 to see performance can be increased.

# Chapter 8

# Bibliography


[1] Padua, D. (2000). The Fortran I Compiler. *Computing in Science & Engineering*, 2(1), 70-75. http://www.cs.fsu.edu/~lacher/courses/CIS49301/notes/cise_v2_i1/fortran.pdf

[2] Cheatham, T. E. (1986). Reusability through program transformations. *Readings in Artificial Intelligence and Software Engineering*, edited by C. Rich and RC Waters, 185-190.

[3] Goldberg, A. T. (1986). Knowledge-based programming: A survey of program design and construction techniques. *Software Engineering, IEEE Transactions on*, (7), 752-768.

[4] Bundy, A., Smaill, A., Wiggins, G. (1990, January). The synthesis of logic programs from inductive proofs. In *Computational Logic* (pp. 135-149). Springer Berlin Heidelberg.

[5] Deville, Y., Lau, K. K. (1994). Logic program synthesis. *The Journal of Logic Programming*, 19, 321-350.

[6] Flener, P., Lau, K. K., Ornaghi, M. (1997, November). Correct-schema-guided synthesis of steadfast programs. In *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference* (pp. 153-160). IEEE.

[7] Basin, D., Deville, Y., Flener, P., Hamfelt, A., Nilsson, J. F. (2004). Synthesis of programs in computational logic. In *Program Development in Computational Logic* (pp. 30-65). Springer Berlin Heidelberg.

[8] Colón, M. A. (2005). Schema-guided synthesis of imperative programs by constraint solving. In *Logic Based Program Synthesis and Transformation* (pp. 166-181). Springer Berlin Heidelberg.

[9] Srivastava, S., Gulwani, S., Foster, J. S. (2010). From Program Verification to Program Synthesis. http://www.cs.umd.edu/~saurabhs/pubs/popl10-syn.html

[10] Perelman, D., Gulwani, S., Grossman, D., Provost, P. (2014). Test-Driven Synthesis. http://research.microsoft.com/en-us/um/people/sumitg/pubs/pldi14-tds.pdf

[11] Potassco, the Potsdam Answer Set Solving Collection. http://potassco.sourceforge.net

[12] Anger, C., Konczak, K., Linke, T., Schaub, T. (2005). A Glimpse of Answer Set Programming. http://www.cs.uni-potsdam.de/wv/pdfformat/ankolisc05.pdf

[13] Corapi, D., Russo, A., Lupu, E. (2011). Inductive Logic Programming in Answer Set Programming. http://ilp11.doc.ic.ac.uk/short_papers/ilp2011_submission_20.pdf

[14] Law, M., Russo, A., Broda, K. (2014). Inductive learning of answer set programs. https://www.doc.ic.ac.uk/~ml1909/ILASP_Paper.pdf