5-1-2014

Meta Concepts: A Knowledge-Based Code Generation System

Nicholas Moss

Nicholas Moss

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

George Luger , Chairperson

Thomas Caudell

Pat McCormick

Meta Concepts: A Knowledge-Based Code Generation System

by

Nick Moss

B.S. in Computer Science with Highest Honors, University of California, Santa Cruz, 2005

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

Spring, 2014

Acknowledgments

This thesis is dedicated to my parents. I greatly thank them for their love and support and have enjoyed living close to them and appreciate their company while I completed this project and as I continue on with graduate school. Special thanks to my thesis advisor Professor George Luger at UNM for his expert help and advice and for finding time to work on this thesis with me. Special thanks to Pat McCormick and the Applied Computer Science Group at Los Alamos National Laboratory for providing an interesting job and income that has helped to get me through graduate school and for giving me an opportunity that has allowed me to develop skills in LLVM/Clang which have been a key ingredient of various aspects of this project. Thanks to Professor Thomas Caudell for his help as a thesis committee member and for his excellence in teaching CS 547 (Neural Networks) at UNM which provided me with a strong background in neural networks and machine learning in general and practice in quantifying and presenting my experimental results. Thanks to Apple Inc. for creating such a great UNIX-based platform that is a joy to work with. Thanks to the LLVM/Clang team for creating the amazingly powerful Clang compiler which is extremely well designed and implemented which I was able to modify in order to create the MML++ language/compiler which is used throughout this project and for compiling the Meta Framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose

People have an amazing ability to solve complex problems by performing a sequence of simpler operations (i.e: functions/procedures which take input variables and produce output variables). We are able to do so even when there exists a large number of possible choices for such operations and when the number of combinatoric ways that these operations can be chained together is astronomical. On the other hand, computers typically do not solve problems this way and have to be programmed with a precise set of instructions. What is it that allows us to perform such a feat while computers cannot? One of the major features that sets us apart from computers is our ability to draw upon our large range of knowledge and its connections to the problem at hand. *Meta Concepts* aims to use knowledge-based information in this way to enable the automated generation of code in order to solve problems in cases where solutions would otherwise be difficult to devise by hand. Meta Concepts is an object-oriented coding system whereby the user specifies and works with an *ontology* of *concepts* or types/classes which captures knowledge about their usage

and metadata about their methods, how method calls can be chained together, and associated method parameters and constraints. By augmenting the code generation process with knowledge-based information, the system is able to significantly narrow and prioritize its search through the otherwise vast search space in order to quickly generate high-performing solutions.

To describe this project in simpler terms: Meta Concepts is a software system which is used to automatically generate solutions to solve complex problems – it is a computer program which creates other computer programs – more specifically it generates a fragment of a program which is useful in the context of a larger program which we have created to solve a well-defined problem. It is applicable in cases where we seek solutions which are a composition of simpler primitives much in the same way that molecules are a composition of atoms which in turn are made up of elementary particles such as protons and electrons or how mathematical equations are made up of simpler operations such as addition or division. These systems are governed by rules or laws which describe how their atomics interact or how they can be combined, e.g: protons attract electrons, a number which represents a length can be added to another length, but an area cannot be added to a length. Meta Concepts allows the user to specify the atomics and the general rules by which these atomics can be potentially combined in order to automatically generate solutions. In order for this process to be automated, the user must provide some way of evaluating the solutions which the system produces and to assign to them some quantitive rank or *fitness*. Evaluation is done either against some data, e.g: an equation which describes some system for which we have collected data, or by judging the performance of the larger program when the generated solution is "plugged" into the larger program e.g: part of a behavior for an agent or simulated robot, how well the generated solution performs or makes decisions when playing a game, etc. In this way, the system can rapidly generate and test several solutions in a randomized or stochastic fashion until it finds a solution which solves our problem.

1.2 Overview

Meta Concepts is an object-oriented coding system whereby the user defines an ontology of concepts and the methods that act upon them. The system takes the idea of strong-typing to a higher level. In typical programming models, we work with types where the compiler knows little more than that a certain type is at the lowest level a floating point number or how a data structure is laid out in memory. In Meta Concepts the code generator has deep knowledge about its types and manipulates variables/concepts such as `Position`, `Mass`, `Length`, etc. that have a specific meaning combined with attributes on these concepts, e.g: whether a concept instance is a vector, a set, a constant, how many times a variable has been used as in input or output, how many calls ago it was produced, and much more. The system makes it easy to add new types and attributes and facilitates extending the method matching criteria. We can set requirements and constraints on methods such as the required inputs and describe what attributes the outputs have. We can also define quantitatively how strong of a match a certain method is based potential inputs and their attributes, thereby expressing hard and soft constraints. The concept's type is the most important cue used by the code generator, and additionally, attributes attached to concept provide further hints to the code generator that enables the highly prioritized search that is at the core of the system's code generation process.

In Meta Concepts, concepts are implemented as C++ classes. I am using a language of my own design called *MML (Meta Modeling Language)* which allows the generated code to be interpreted by interfacing with the C++ side that performs the actual execution. Additionally, I have created a modified version of the Clang C++ compiler to create the *MML++ language*, a superset of C++. MML++ adds several powerful features used by the system, among them: the ability to define metadata directly in the C++ class header files in order to describe knowledge-based information about methods and their inputs/outputs/return values.

A generated solution is *simply* (this is said with some irony due to the combinatoric nature of the problem) a chain of method calls. An important design feature of the system is that it does not generate control flow but rather relies entirely on methods defined in the ontology. For example, rather than generate a for loop that performs a computation such as a sum or an average, a method is defined that performs such functionality. Another important strength of the system is that the underlying implementation of concepts is written in C++ and can be arbitrarily complex – we are defining an interface to the classes which allows them to be manipulated in a meaningful way which is more likely to produce good results than if we were simply performing random calls.

The following example output in MML depicts a final solution from a simple program which is subsequently referred to as the `setops` example/test case:

```
Length* length_0 = b.setComplement(f);
length_0.foo();
Length* length_1 = a.setComplement(d);
Length* length_2 = length_0.setComplement(length_1);
Length* length_3 = a.setUnion(length_2);
Length* length_4 = f.setUnion(d);
Length* length_5 = length_4.setIntersection(length_3);
g.setObj(length_5);
```

Figure 1.1: Sample solution

The user provides as input four concepts of type `Length` named `a`, `b`, `d`, and `f`, and seeks as output a concept of type `Length` named `g`. In the process, the code generator creates a few temporaries named: `length_0`, `length_1`, etc.

The following excerpt of MML++ code depicts how metadata is defined in the `Real.h` header (concept `Length` derives from the `Real` concept) in order to show how method metadata is defined for some of the methods used in the example above:

```
 /*[
  self: [set:true, vec:true],
  c: [set:true, vec:true],
  ret: [set:true, vec:true]
]*/
Concept* setUnion(const Concept* c) const;

/*[
  self: [set:true, vec:true],
  c: [set:true, vec:true],
  ret: [set:true, vec:true]
]*/
Concept* setIntersection(const Concept* c) const;
```

Figure 1.2: Sample methods and metadata

The final generated metadata picks up certain attributes already provided by the C++ syntax such as whether a parameter, the object called upon (the "this" object), or the return value is `const`. It also knows that because we have used the generic `Concept` type that the method will polymorphically produce or take as input a `Length` when it is called on a `Length` type. In this way, rather than define a method, say `setUnion()` for each concept that extends `Real`, we can simply define it once.

To solve a problem with Meta Concepts, the user begins by specifying the problem inputs and desired output concepts and by describing these variables with as many relevant attributes as possible. We are not concerned with generating a complete program but rather a *module*, or section of code which is generated then "plugged" into a larger *host program* for evaluation. The user devises some way of evaluating the overall fitness of the program instance when the module is plugged into the program and evaluated.

In the appendices, under the Sample Program Setup section, we list the source

code for the setup and general control flow of this program as it interfaces with the Meta Concepts system. This problem is a very simple and contrived example to illustrate the basics of how the system works. In this case, knew what the final solution was and we calculated the error directly, but imagine a more sophisticated program in which we were generating a section of code where we do not know the error *a priori* but the module serves, for example, as some sort of control mechanism. Here we might be aiming to evolve a complex behavior in the host program and we calculate the fitness by how well the larger program performs, not its error.

Chapter 2

Related Work

In this chapter I will briefly describe two existing prominent fields of work within computer science, both active areas of research in artificial intelligence, which are relevant to Meta Concepts: genetic programming and constraint programming.

2.1 Genetic Programming

Genetic programming (GP) is an evolutionary-inspired automated code generation technique which is a special form of *genetic algorithms (GA's)*. The roots of GP date back to 1950's, with related work done by Fogel, Holland, Barricelli, Cramer, and Rechenberg during the 1950's to 1980's. However, GP as it is known today, was developed and formalized by John R. Koza in the 1990's. [1]

2.1.1 Genetic Algorithms

Genetic algorithms are a machine learning technique that have been successfully used to solve a large range of problems involving optimization and search. In genetic

algorithms we encode solutions as chromosomes. In typical form, the chromosome consists of a binary string. Key to GA, is the crossover operation which involves selecting two solutions A and B stochastically from a population in order to create a new solution by recombining bits of A and B. The simplest crossover approach is to pick a random crossover point and create a new solution C which is formed by splicing together the A's portion of the chromosome to the left of the crossover point and B's to the right. Mutation is also employed, by randomly modifying parts of the chromosome, but usually applied at a lower rate and as secondary to crossover.

After mutation and crossover, the new solution is evaluated and perhaps replaces a solution of lesser fitness in the population. By iteratively applying mutation and crossover, the population moves towards a state of higher fitness. We stop after a fixed number of iterations, after some convergence criteria is achieved, or after finding some candidate solution which achieves a desired fitness.

Genetic algorithms are typically easy to implement but it has been difficult to grasp and formulate theoretically how and why they succeed. The building block hypothesis attempts to describe their success. Goldberg describes the building block hypothesis heuristic as follows:

> Short, low order, and highly fit schemata are sampled, recombined and resampled to form strings of potentially higher fitness. In a way, by working with these particular schemata (the building blocks), we have reduced the complexity of our problem; instead of building high-performance strings by trying every conceivable combination, we construct better and better strings from the best partial solutions of past samplings... Because highly fit schemata of low defining length and low order play such an important role in the action of genetic algorithms, we have already given them a special name: building blocks. Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a

> genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks. [2]

2.1.2 Genetic Programming Fundamentals

Genetic programming is a special form of GA where the chromosomes we are manipulating are pieces of executable code. A large variety of methods and approaches have been utilized to perform genetic programming. In this section I will describe GP in its most basic and traditional form. GP is typically performed by representing code fragments in tree form. For example, the code expression `a * (b - c) + d` can be represented in tree form as:

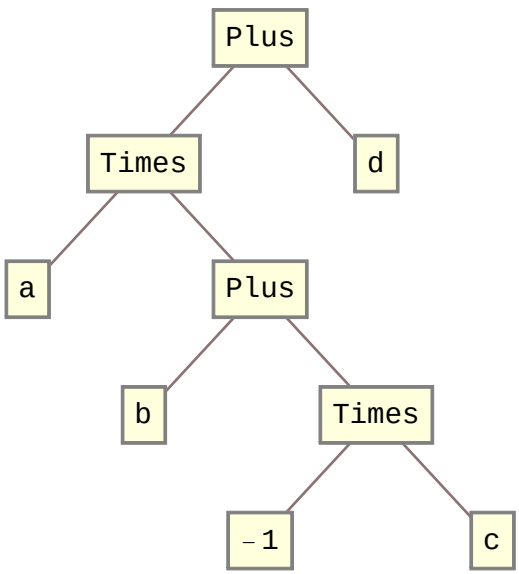


Figure 2.1: Sample GP tree

The *primitive set* consists of the set of terminals and non-terminal functions used by the GP run. The terminal set is made up of input variables, constants,

or functions of arity 0 – those that reside at the leaf or terminal nodes of the tree. So in our example above, the terminal set $T = \{a, b, c, d, -1\}$ and the function set $F = \{Plus, Times\}$ constitute the primitive set. Usually the arity of the functions in F is known for each function beforehand. Here we have restricted ourselves to using $arity = 2$ functions or binary functions.

2.1.3 Initial Population

GP runs are usually performed with a limited population size. Typical population sizes for small problems might be on the order of thousands. The ideal population size usually depends on the nature of the problem we are attempting and often we can get by with a small population size for simpler runs. Large populations entail greater memory footprints and can take longer to generate the initial population and require higher processing and convergence times. Once a reasonable population size has been chosen, GP runs employ different strategies for forming the initial population. The `grow` method and the `full` method are two such approaches to generating the initial population. [9] In both methods we are generating solutions up to some desired tree depth d_m, the proper setting of which depends on the complexity of the target solution. Note that the complexity of the solution grows exponentially with d_m. If we are using strictly binary functions then the number of nodes in a tree of depth d_m is $O(2^{d_m})$.

In the `full` method, we form subtrees by picking random functions from F until depth d_m is reached, at which point, we pick terminals randomly from T to complete the tree. The `grow` method, on the other hand, allows for the creation of trees whose size and shape is variable. Here, we randomly choose from both T and F until depth d_m is reached, at which point we only choose terminals. Often it is useful to apply a hybrid approach in generating the initial population, e.g: generate half of the population with the `full` and the other half using `grow`.

2.1.4 Problem Setup, Training Data, and Fitness Evaluation

Consider the following scenario, which is one common usage of GP: we wish to perform a *symbolic regression* whereby we would like to generate a function, e.g: $y = f(x)$. We do not know the exact form of the function but we have a hunch that it can be created as a composition of a small number of arithmetic functions such as *Add* and $Times$ and a limited number of provided terminals. Note that the function does not necessarily have to be linear (one of the powerful features of GP is that the function could be anything, involving any types of primitive functions we wish) – we could just have well as included a Pow function for raising a value to a power, for instance.

We have collected data which maps input values of x to output values y. So we have training data which consists of the sets $X = \{x_1, x_2, ... \ x_n\}$ and $Y = \{y_1, y_2, ... \ y_n\}$. We generate successive attempts at finding a function that fits the data, $g_i(x)$. Then we can use the *root mean square (RMS)* method to derive an aggregate fitness measure as follows:

$$RMS_i = \sqrt{\frac{\sum_{j=1}^{n} (y_j - g_i(x_j))^2}{n}} \tag{2.1}$$

The GP process can then calculate a fitness $f_i = 1/RMS_i$ for each generated function $g_i(x)$ by applying $g_i(x)$ over the the training data. The exact value of the fitness measure does not matter, it is only important that fitness values are relative to one another according to certain desired features that allow us to compare the relative correctness or utility of the generated functions.

2.1.5 Selection, Mutation, Crossover, and Replacement

After the initial population has been formed, in this simple scenario, the solutions are evaluated against training data as described using a method such as RMS. We iteratively proceed by stochastically selecting solutions from our population. There are a variety of methods that may be employed to select a solution from the population. We can select one uniformly by picking a random solution from the population, but it is sometimes more effective to pick a solution according to some distribution based on solution fitness so that we are more likely to select fitter solutions. Mutation and crossover are the two main operations that are then applied to the solution(s) selected.

In mutation, we select a solution, and somehow alter it, usually in some minor fashion. A wide variety of mutation techniques may be applied, for example, we may simply pick a node in the the tree as a mutation point, erase the subtree at that node, then regenerate that portion of the subtree using the `grow` or `full` method. Once a solution has been mutated, we reevaluate its fitness. We have a few choices on how to enter it into the population. One approach is to simply enter the new solution into the population at the end of the list. More commonly, when the solution is at its capacity, we would replace a solution only if its fitness was higher than that of the lowest fitness solution.

GP, like GA's in general, derives it power from the crossover operation. So typically we employ some kind of rate to determine whether we do mutation or crossover. A typical heuristic is to do mutation roughly 5 percent of the time and crossover for the rest. Crossover works by selecting two solutions A and B stochastically as described above for mutation. There exists a large variety of crossover approaches to choose from. One of the simplest approaches involves selecting two random crossover points p_1 from A and p_2 from B to form a new solution C by removing the subtree

in A rooted at p_1, replacing it with the subtree in B at p_2. After crossover has been performed, we evaluate the new solution and possible enter it into the population – the same as we did for a mutated solution.

2.1.6 Stopping Criteria

If the run is successful, we have found a solution that achieves some desired fitness f_d after a number of iterations and we are done. If we are not successful, then the run can become stagnant as the population of solutions converges to some local optimum. For this reason, it uncommon for an individual GP run to take a significantly long time – either we will find a solution or the run will stagnate and will not make much further progress. For this reason, as the run progresses it can be helpful to monitor: the best, average, and worst fitness. We can also apply certain convergence heuristics to these measures in order to automatically halt the run. We might attempt several more runs if we cannot find a solution or alter various parameters and approaches such as:

- population size
- initial solution generation approach / parameters
- mutation and crossover rate
- mutation and crossover approaches
- modifying the primitive set

2.2 Constraint Programming

> A constraint is a restriction on a space of possibilities; it is a piece of knowledge that narrows the scope of this space. Because constraints arise naturally in most areas of human endeavor, they are the most general means of formulating regularities that govern our computational, physical, biological, and social worlds. Some examples: the angles of a triangle must sum to 180 degrees; the four nucleotides that make up DNA strands can only combine in particular sequences; the sum of the currents flowing into a node must equal zero; Susan cannot be married to both John and Bill at the same time. Although observable in diverse disciplines, they all share on feature in common: they identify the impossible, narrow down the realm of possibilities, and thus permit us to focus more effectively on the possible. [3]

The field of *constraint programming (CP)* originated from the need to embed constraints in logic programming languages such as Prolog. [4] The types of problems CP is concerned with also encompasses a more restricted form of CP that AI researchers were actively concerned with in the 1970's, research of which that has benefited CP and CP solvers greatly. [5] CP is a technique used for solving optimization and search problems and has grown to be a much broader field that is used today to solve complex problems in a variety of application domains including [5]:

- the construction of efficient parsers for natural language processing
- computer algebra systems for solving systems of equations and for expression simplification
- electrical engineering including circuit design

- operations research in order to solve various optimization and scheduling problems
- molecular biology including DNA sequencing

2.3 Formal Description

A *constraint satisfaction problem (CSP)* is defined over a set of variables $V = \{x_1, x_2, ... \ x_n\}$. where each variable can take on values over some known domain, i.e: $x_1 \in D_1, \ x_2 \in D_2, ... \ x_n \in D_n$. $D = \{D_1, D_2, ... \ D_n\}$ where each domain may be finite or infinite. The constraints $C = \{C_1, C_2, ... \ C_n\}$ are relations that hold between variables over their domains. For a solution to satisfy the CSP, it must simultaneously satisfy all constraints or it must be true that $C_1 \wedge C_2 \ ... \wedge C_n$.

For example, consider the following simple CSP involving variables $x_1, x_2 \in \mathbb{Z}^+$ with the following constraints:

- $x_1 > 10$
- $x_2 < 5$
- $x_1 + x_2 = 13$

Then the following pairs (x_1, x_2) satisfy this CSP: $(11, 2)$, $(12, 1)$

A common usage of CPS's is to formulate the CSP as an optimization problem whereby an objective function is specified which we seek to minimize or maximize. If we extended our example above with the maximization of the objective function $f(x_1, x_2) = x_1^2 + x_2$, for instance, then our final solution becomes: $(12, 1)$.

Solving a CSP is, in the general case, an NP-Hard problem. Even in a very simple type of CSP, called SAT, where we restrict our variables b_i over the boolean

domain, and represent clauses in ternary CNF (conjunctive normal form) [1], e.g: $C = (b_1 \vee b_2 \vee b_2) \wedge (b_3 \vee b_4 \vee b_5)$ ... , then this problem is NP-complete and unless P = NP, there is no algorithm that runs in polynomial time that can solve SAT in polynomial time in the number of n variables. [6].

2.4 CSP Solvers

While CSP solving is an NP-Hard problem in the general case, often we are concerned with CSP's containing a relatively small number of variables and clauses and for those in which the overall complexity is difficult but tractable. A variety of domain-specific and general methods can be applied to solve many CSP's of interest. Domain specific methods are usually provided in the form of packages related to a particular CSP to be solved. Examples of domain specific methods include [5]:

- solvers for linear systems of equations
- linear programming packages
- implementation of the unification algorithm, i.e: in automated theorem proving

General methods are concerned with generic methods for traversing and reducing the search spaces encountered in common across different CSP's. A thorough description of the methods involved in solving CSP's is beyond the scope of this thesis, but solvers typically employ advanced artificial intelligence techniques such as tree traversal and search, rule-based frameworks, mathematical techniques such as the simplex algorithm, solving systems of linear equations, and methods specific to CP

[1]Note that binary SAT CNF clauses (2-SAT) are limited in expressive power, and solving them is in P, but in going to ternary CNF clauses (3-SAT) which are equivalent in expressive power to any k-SAT for $k \geq 3$, k-SAT is NP-Complete.

such as the conversion of implicit constraints to explicit constraints and constraint propagation. [5]

2.5 Constraint Programming Languages

> ... consider Ohm's Law: $V = I \times R$ which describes the relationship between voltage V, current I and resistance R in a resistor. In a traditional programming language, the programmer cannot use this relationship directly, instead it must be encoded into statements which compute a value for one of the variables given values of the other two variables... Requiring the programmer to explicitly maintain relationships between objects in the program is reasonable for applications in which the relationships are simple and static. However, for many applications, the crux of the problem is to model the relationships and find objects that satisfy them. For this reason, since the late 1960's, there has been interest in programming languages which allow the programmer simply to state relationships between objects. It is the role of the underlying implementation to ensure that these relationships or "constraints" are maintained. Such languages are called *constraint programming* languages. [7]

As described above, a *constraint programming language (CPL)* is a programming language which includes first-class programming constructs for working with CP and solving CSP's. Early CPL systems were done is a rather ad-hoc fashion allowing existing languages to be extended so that users could benefit from CP in a limited fashion. For example, the CPL could track relations between variables and could then deduce certain unknowns from CP relations given other known variables.

More recently, there has been a renewed interest in CPL's which include a full integration of CP capabilities. [7] One of the powerful features of a CPL is that

once we have set up a certain problem as a CSP, multiple queries can be processed, yielding answers to a variety of questions whereas in traditional programming, we would have to rewrite certain relations specifically tailored to answer each question. Consider the example for Ohm's law that was given in the passage about but imagine a more involved equation involving several more variables – the equation itself is a constraint. When we are faced with the task of performing a query, certain variables may be unknown, but we can limit the scope by specifying additional constraints specific to a query. We can then answer a number of questions:

- does a solution exist at all given certain additional constraints?
- what is the effect on the solution of modifying, removing, or adding another constraint?
- what is the solution or solutions which satisfy the constraints?
- among multiple solutions, what is the solution that maximizes some objective function?

One of the advantages of CSP's, especially when combined with CPL's, is that they are a high-level description or type of declarative statement or assertion in which the user of a CSP system focuses on the *what* vs. the *how*, typically specifying the CSP in a very natural mathematical language. [7] Despite the high-level nature of CPL's, there will always be a certain skill and learning curve involved with the proper formulation of constraints and often it is vital to possess knowledge, at some level, of how the solver process works as well as an understanding of the system's limits as it is easy to devise CSP's which are beyond the capability of the solver but which could possibly be refactored to become tractable.

Despite the utility of CPL's, one does not necessarily have to use a full CPL to gain the benefit of CP capabilities. There exists a large number of high-quality and

freely available CSP solvers than can be called from mainstream languages such as C/C++ as listed on the Wikipedia entry for constraint programming [8].

Chapter 3

Motivation

The end goals, requirements, and design features of the Meta Concepts system are summarized as follows:

- To design and implement a general-purpose software system which will scale to being capable of solving complex problems in a variety of domains including the development of novel and human-competitive solutions in various areas such as mathematical and scientific systems, especially AI systems, e.g: components of agent behaviors and control mechanisms, inference, decision-making, etc. The solutions generated are not complete programs but components which are useful in the context of a larger system.
- The system shall be high performance both in its design and algorithms and in implementation and should ultimately be capable of running on concurrent and parallel systems.
- The system shall be extensible so that it is easy to add new concepts, methods, and matching criteria and constraints.
- The system shall enable an informed search process so that it takes advantage

of both the user's expertise and domain knowledge as well the system's ability to quickly generate and evaluate solutions. Later versions of the system may also allow the user to control various aspects of the code generation process, e.g: flag interesting solutions as higher priority.

- Facilitate the creation of general-purpose and highly-reusable ontologies of concepts.

- Enable the system/ontology to learn as it applied to more and more problems.

- Allow the system to be run in server mode. For example, a long run could be initiated on a remote machine and client systems can connect to that run in order to monitor or control its progress. The system includes a network-capable GUI client program for this purpose.

- Support analysis of the operation of the code generation process so that debugging, diagnosis when difficulties are encountered, and further development and extension of the code generation system is made easier.

- The system shall be robust in that no generated solution shall cause the system to halt indefinitely or crash. This is accomplished via extensive use of exceptions in the framework and supporting code and by enabling the evaluation engine to be interruptible so that it can be terminated if a user-specified timeout threshold per solution instance is exceeded.

- Facilitate complex implementations of concepts by basing the system on a mainstream language: C++. Allow generated solutions to be easily integrated into a C++ program.

- Provide a universal and generic fitness evaluation approach that supports both supervised and unsupervised learning.

- Make it easy to add new concepts and methods which can be interfaced with the code generator by allowing Meta Concepts comments/data to be specified next to their definition in their C++ header file. Generate metadata from existing C++ syntax as much as possible, e.g: the C++ `const` keyword. Perform extensive error checking as early as possible to detect and report problems related to Meta Concepts metadata to ensure the correctness of the ontology as much as possible.

3.1 Relations to Genetic Programming

Meta Concepts was inspired, in part, by genetic programming and earlier related work and research I conducted independently which led to my discovery of GP as a field which could be learned from. Genetic programming and Meta Concepts both have similar goals: they both aim to automate the generation of code using stochastic methods and rule-based frameworks, solving some problem which generates solutions which are evaluated with a fitness metric.

Several advanced variations on the basic GP scheme presented in the previous section have been created, some of which aim to impart the GP process with additional information and constraints that guide the code generation process such as the role that knowledge-based information serves in Meta Concepts. One such mainstream GP variation, uses grammar-based constraints to express relations where the non-terminal rules are described by productions similar to those found in context-free grammars. Other advanced variations of GP are able to generate sequences of assembly code; others use stack-based machines and alternative computing models; while others have the capability of generating and accumulating modules of reusable code. [9] There is an incredible amount of interesting and fruitful prior work that has been done in GP. As such, I will not attempt to compare the work of Meta Concepts

to GP in the general case, but will use the *traditional basic genetic programming approach ("BGP")* as presented in the sections prior as a baseline to discuss and highlight some of the value-added features that Meta Concepts incorporates.

One of the primary features that sets Meta Concepts apart from BGP is the use of knowledge-based information to both guide and constrain the code generation process. In this sense, we are utilizing both the best of the user's knowledge combined with the power and speed of the computer and Meta Concepts system. This knowledge is provided primarily by strongly-typed concepts and metadata attributes. In the BGP example provided above, we have specified two non-terminal functions, *Add* and $Times$. When we create the initial population the only restriction provided in the generation of trees involving these functions is that they take two arguments. The system has no notion of the underlying concepts, or semantics of the values it is manipulating. Therefore it is likely to generate several trees which do not have valid semantic meaning. For example, suppose we were attempting to devise a function which computes the area of a regular n-sided polygon as a function of the number of the n and the length of each side l (this problem is presented later in the Results section). In Meta Concepts, we can specify, rules, relations, and constraints that describe the valid operations that can be performed, e.g:

- A length can be added to another length, but a length cannot be added to an area.
- A length can be divided by another length to create a length ratio.
- An angle can have the function $sin()$ or $cos()$ applied to it which produces a length ratio.
- A concept of type C can be multiplied by a C ratio to produce a C.
- A length can be multiplied by another length to produce an area.

- π is a static constant which is available for use whose type is an area ratio.

In BGP, on the other hand, we generate the initial population, and iteratively apply crossover or mutation which could produce several meaningless expressions when the variables we are manipulating are assigned types, e.g: multiplying an area by another area, adding an angle to a length, and so on. Not only do constraints like the above narrow the search space, but they ensure the validity and correctness of solutions. Furthermore, Meta Concept ontologies are highly reusable. Once we have defined an ontology that defines the above concepts and relations for length, area, angles, etc. we can then put it to use to solve several different problems.

Meta Concepts does not perform mutation or crossover but provides analogous operations. For the same reasons that generating the initial population is likely to generate several meaningless expressions, the same applies to crossover and mutation, furthermore both are destructive. Here, I argue that the Meta Concepts analogs are more likely to produce stronger results. In Meta Concepts, the analog of mutation is accomplished by re-queueing a solution to the code generator. The code generation process can use this solution as a starting point to continue generating code from that solution then it may finally make a different assignment from temporaries to the final desired outputs. In this way, we have given the code generator an opportunity to alter or mutate an already high-performing solution without erasing any of its code which may be vital to the final solution.

The Meta Concepts analog of GP crossover is the *merge* operation which will be described more completely in subsequent sections. Merge works by selecting two solutions A and B so that their state can be merged, then queues the merged solution C so that the code generator can further modify it, the same as in the case of mutate. Again, merge is non-destructive and is more likely to generate useful results because the same constraints that were applied in generating A and B are applied to C which guarantees correctness so long as the ontology is consistent. It depends exactly on

the problem, but in general, the code generator has far fewer choices (many of which are not helpful in creating a solution of higher fitness) to make compared to in the case of BGP crossover where we must pick two crossover points in order to splice together a new tree.

Like BGP, Meta Concepts also suffers from the problem of convergence to local optimums. We are working with a population whose size is much smaller than the overall search space. When we begin crossing over or merging solutions, the population tends to become homogenous over time to the point where the system will either find a solution after some finite amount of time or will have to reset so that we can try again. Meta Concepts includes the option of resetting its state, typically done after some chosen number of iterations if no solution has been found.

Finally, one of the major strengths of Meta Concepts is that it is an object oriented-system and interfaces with a main-stream language: C++. Therefore the implementation can be arbitrarily complex and the solutions we generate can be used in the context of a larger real-world program. We are not limited to mathematical problems such as the symbolic regression scenario that was described earlier. The generated code is a sequence of object-oriented calls on objects thereby the behavior of the final solution can be replicated simply by pasting in the output code of the solution as C++ and linking against the ontology library. In addition, the system does not attempt to generate control flow which is error-prone and causes numerous problems, e.g: infinite loops. Instead, the system relies strictly on methods defined in the ontology as building blocks. This design decision makes the user's task of specifying the ontology and problem instances clearer and avoids certain difficult code generation problems.

3.2 Relations to Constraint Programming

Interestingly, the overall problem of code generation that Meta Concepts aims to tackle can be formulated as a constraint satisfaction problem (albeit a rather complex one) with an objective function where we aim to maximize fitness over some problem instance. The variables are the sequence of specific calls that are performed in a solution as well as the temporaries created and other inputs which are mapped to parameters of further calls and the domain consists of those variables chosen as concepts from the ontology and the methods that the ontology provides. Meta Concepts, in a sense, is a specialized type of CSP solver.

More tangibly and useful, on the other hand, is to consider the ways that Meta Concepts performs matching between call sequences and matching candidate variables and call parameters and to understand how this is a type of CSP that could benefit from the theory of CP and perhaps existing CSP solvers.

One of the approaches that was attempted early on that was abandoned because it was not tractable was: when choosing which variables in the state to map to a call's parameters, to generate the permutations over all possible variable to parameter mappings. Not only can this lead to a large number of permutations when the number of call parameters is large, but of more concern, is that when solution size increases, we have more temporaries to choose from, then generating the permutations explicitly becomes infeasible.

Unfortunately, the problem is that this makes it difficult to test conditions that involve constraints over multiple parameters, therefore the current approach is that the only constraints that may be applied are those involving the potential variable to be mapped to a certain parameter in isolation. In this way, after a method has been selected, we generate the candidate variables that can be mapped to a certain parameter, filtering out those which violate the *hard constraints*. A hard

constraint is one that causes the `match()` method to return $m = -1$ as described in subsequent sections. A *soft constraint* is expressed by mapping to candidate to some positive value for m so that the candidates can be chosen stochastically as linearly proportional to m. For example m might partially be a function of how many calls ago a variable was produced, preferring more recently produced variables, whereas considerations of type and whether attributes match (e.g: both parameter and variable are a vector) are expressed as hard constraints.

Some of the higher level goals of Meta Concepts are directly inline with the aims of CP, especially CPL's. Namely, that we wish to express relations and constraints between variables and parameters, especially involving type and attribute domains. In the case of Meta Concepts, the goal is so that several different variations of these relations can be generated as potential solutions whereby we constrain the solution search space and ensure the validity of generated solutions.

Chapter 4

Approach

4.1 Supporting Framework and Languages

A large part of the implementation of this project relies upon the foundation of the *Meta Framework and Languages* which I have been developing independently since 2003. In this section I will briefly outline specific features of the Meta Framework which are utilized by this project.

4.1.1 The M Language and Interpretation

The Meta Framework's *M language* is one of the most essential foundations on which Meta Concepts is built. It allows code to be represented and manipulated as data. M code can be created at runtime and executed through any `MObject` subclass. The `mnode` type is used recursively to represent M code. In the C++ code segment below, we construct M code corresponding to a functional node `f = Add(x,1)` and run it through an interpreter.

M can be thought of as an interpretable AST. Besides, interpretation it is also

```
MObject object;
mnode d = mfunc("Def") + msym("x") + mnode(2);
object.process(d);
mnode f = mfunc("Add") + msym("x") + mnode(1);
mnode r = object.process(f);
cout << "r is: " << r << endl;
```

Figure 4.1: Example of how M code is created and executed programmatically

used throughout the framework when generating compiled code.

MML

MML (Meta Modeling Language) is a programming language, originally designed for modeling and simulation, built on top of the the M foundation. M excels at manipulating or constructing code at runtime, but is difficult to read or write larger programs in. MML provides a solution by defining a language that gets parsed to M code which can then be directly interpreted by an `MObject`. Thus, MML can be thought of as a convenient front-end to M code and M-based interpreters. MML is an easy-to-use, semi-weakly-typed language which is based on the syntax of C/C++. The following MML code is the equivalent to the M code above.

```
r = x + 1;
```

Because of M's functional programming roots, we can easily define, manipulate, and execute closures, e.g:

```
f = {x + 1};
f.push({y}); // f is now, in M: Add(x,1,y)
x = 2;
y = 9;
eval(f); // evaluate f
```

Figure 4.2: Sample MML code

4.1.2 MProc and MProcTask

Another important feature of the Meta framework used by this project is a graph-based concurrency system provided via the `MProc` and `MProcTask` classes. `MProc`'s are connected to eachother to form a network or task where chaining order determines execution sequence and whereby parallel execution paths can be formed or joined with other paths. The semantics of the `MProc` system are highly configurable allowing the creation of complex networks, that, for the purposes of this project, enable various search methods to be defined and run in parallel.

`MProc`'s signal each other with the `mvar` data type – an essential type of the Meta framework which is capable of representing virtually any type of data recursively. The signals provide a mapping that the `MProc` uses to determine whether or not to activate when signaled and also allows state to be passed, enabling the passage of data to be entirely localized, if desired, such that state propagation through signals allows several instances of the `MProc` to be run unhindered in parallel.

`MProc`'s once activated, are queued to an associated `MProcTask`. `MProcTask`'s coordinate a specific processing goal, providing high-level thread-pooling and queueing with optional application-specific priority of queued items.

4.2 Concepts, Attributes, and Matching

Each concept in the system derives from the `Concept` base class which in turn is derived from `MObject`. Subclasses of `MObject` are then capable of interpreting M code and in MML++ they receive special benefits, namely that all of their M-compatible methods implemented in C++ are then callable by the interpreter. Being M-compatible entails that the method's return value and parameters can be encoded as an `mvar` or `mnode`.

Because defining and modifying method signatures is such a frequent task, Meta Concepts makes the specification of metadata as convenient as possible, allowing metadata definitions to reside right next to their method definition in the corresponding C++/MML++ header file. Metadata specifications are also checked for consistency against their method definition. Methods which include a Meta Concepts comment just before their definition, are then made available to the code generator.

```
 /*[
  self: [set:true, vec:true],
  c: [set:true, vec:true],
  ret: [set:true, vec:true]
]*/
Concept* setUnion(const Concept* c) const;
```

Figure 4.3: Sample method and metadata

Meta Concepts comments can also be used to describe the concept class itself. Meta Concepts comments are structured as key-value specifications where each key either refers to a parameter, the return value with the `ret` keyword, or the "this" object with the `self` keyword. The Meta Concepts comment is itself an `mvar` where each contained keyed value consists of an `mvar` map of the attributes to be set, e.g: `[set:true, vec:true]`.

During metadata parsing, when a syntax error is detected, the ontology will output an error message indicating exactly where the error occurred. Attributes are set via mutator methods on the `Concept` or subclass, also reporting a similar error when an invalid attribute or incompatible attribute value is given.

In the common case, each keyed value describes either the input requirements and matching criterion when a variable is applied to a parameter or the output state of the specified parameter when it produces a variable (e.g: a return value). However in less common cases, such as when a parameter is both an input and an output (a parameter which is modified), we may wish to describe its output state as being distinct with the `_out` key extension. In this case, the attribute has its out state merged with the input state (giving precedence to values in its out state). Consider the following method `foo()` which takes an input `x` which is also modified by the method call. Similarly, we could create a method that modifies the "this" object by dropping the const and possibly specifying its out state.

```
/*[
    self: [set:true, vec:true],
    x: [set:true, vec:true],
    x_out: [set:false, vec:true]
]*/
void foo(Concept* x) const;
```

Figure 4.4: Sample method and metadata describing a parameter output state

```
/*[
    self: [set:true, vec:true],
    self_out: [set:false, vec:true],
]*/
void bar();
```

Figure 4.5: Sample method where the "this" object is modified

Meta Concepts incorporates a highly extensible call matching system, making it easy to add new attributes and matching criteria. Attributes are applicable to a concept, method, parameter, or variable, or a combination of the four. Currently, the system supports the following builtin attributes:

- `in` - true if a parameter can take as input another variable or true if a variable in the code generation process can be used as an input. A return value is never an input and some parameters may be strictly outputs in the sense that they are produced by the method call. By default every method parameter has `in` set unless overridden.
- `out` - true if a parameter or the "this" object is modified by the method call. For parameters and the return value, the `out` attribute is automatically set from the C++ `const` keyword.

- `const` - Automatically set for parameters from the C++ `const` keyword. A variable in the code generation process has the `const` attribute set if it cannot be further modified, i.e: if it can't be passed as a parameter where the `out` attribute is set.
- `vec` - true if the variable is a vector or a parameter which accepts or produces a vector. The recommended approach to defining the ontology is to utilize the `vec` attribute rather than creating a distinct type for vectors, e.g: don't create concepts such as `PositionVec`.
- `set` - similar in usage to `vec`, `set` is true if the variable or parameter is an unordered set of items.
- `length` - if the type is a vector and the size is fixed, this attribute should be set to indicate its length.
- `poly` - true if the parameter is polymorphic in the sense that it is of the same type as "this". A polymorphic type uses the `Concept` type in its call interface.
- `given` - true if the variable was given as an input as opposed to being that which was created, i.e: a temporary that was produced during the code generation process.
- `age` - the number of calls that have been generated since the variable was produced. Given variables have a fixed age of 0.
- `inputUses` - the number of times a variable has been used as an input.
- `outputUses` - the number of times a variable has been used as an output or modified.
- `enabled` - true if the variable, method, or concept can be used by the code generator or in the case of a variable, further used by the code generation process.

- `takeThis` - true if the parameter can take the same value as the "this" object. The default is false.
- `cloneOf` - the name of a variable that was cloned to create a temporary. This attribute is used by the code generator to initially create a copy of non-const objects so they can be assigned to final outputs in order to support merging.
- `takeCloneOf` - only allow the parameter to accept a specified `cloneOf` object.
- `description` - a textual description of a concept, method, parameter, or variable.
- `remove` - a class-level attribute that allows inherited methods to be removed from a concept and not used during the code generation process.
- `static` - applies only to variables or parameters which, for example, can be designated to only accept a static variable. Static variables are special variables which can be collapsed into a constant which are then embedded directly in the solution.
- `weight` - weight is a recently added attribute that applies to variables and parameters. The weight is a multiplier on the match of a variable whose default value is 1.0. We specify a weight, to give a higher precedence to input variables, for instance, say if we want to make it more likely that a certain variable will be used in a solution.

Certain attributes which are applied to parameters, e.g: `vec` can be set to the `undef` value which indicates that they could accept a variable as input with that attribute set to any value, e.g: `true` or `false`. Boolean valued attributes such as `vec` are assumed as `false` when such an attribute/key is not present.

Additionally, the system supports *post conditions* which are specified using the MML language embedded within the Meta Concepts comment. In the example

below, we use undefined attributes and a post condition to specify that the return value's `vec` attribute depends on whether the input parameter `c` or `self` is a vector.

```
/*[
  self: [vec:undef, ratio:undef],
  r: [ratio:true],
  post: {
    if(r.hasKey("vec") || self.hasKey("vec")){
      ret.vec = true;
    }
  }
]*/
Concept* mulRatio(const Concept* r) const;
```

Figure 4.6: Sample method which uses a post condition

The `Concept` class provides the `match()` method which defines the builtin matching criteria via a virtual method – derived classes can intercept it and define additional concept-specific matching. The `match()` method returns a floating point value which indicates how strong a match is given a parameter and a potential input variable. The figure below depicts an excerpt of how matching works.

A "full" match is performed during the actual code generation process whereas a "partial" match is performed when connecting methods to create the code generation graph (as described in later sections) when less information is available. The match method returns a negative value if no match is possible otherwise higher values are more likely to match a variable with a parameter. In this way, we can express *hard constraints*, by returning a negative value in cases where hard constraints are violated as well as *soft constraints* by returning higher values of m to represent better matches.

```
double Concept::match(const mvar& v, bool full){
  if(attrs_.out && !v.out){
    return -1;
  }
  if(attrs_.vec != v.vec){
    return -1;
  }
  // for brevity, code omitted here
  double m = pow((100.0 - v.age)/50.0, 2.0);
  return m;
}
```

Figure 4.7: The `match()` method

4.3 Code Generation

4.3.1 Initialization

When the ontology singleton is first retrieved, it performs an initialization phase which creates a prototypical concept for each concept subclass. The concept metadata is processed and attached to the concept prototype. An internal `Method` object is created for each method over all concepts and each method is assigned a unique id. The metadata for each method is processed and associated with the method. Extensive error checking is performed so that errors are detected and reported as much as possible before code generation begins.

4.3.2 Code Generation Graph (CGG)

When the user instantiates and initializes a `CodeGen` object by calling the `generate()` method, another phases of initialization occurs. Now the user has specified the input variables and their associated attributes as well as the desired outputs. At this

point, a *Code Generation Graph (CGG)* is created and associated with the `CodeGen` object. The CGG is a cyclic directed graph which connects methods and determines, partially[1], the potential call paths that the code generation process can follow. The CGG is implemented using the Meta Framework `MProc` system as described earlier, where each node is implemented as an `MProc` subclass.

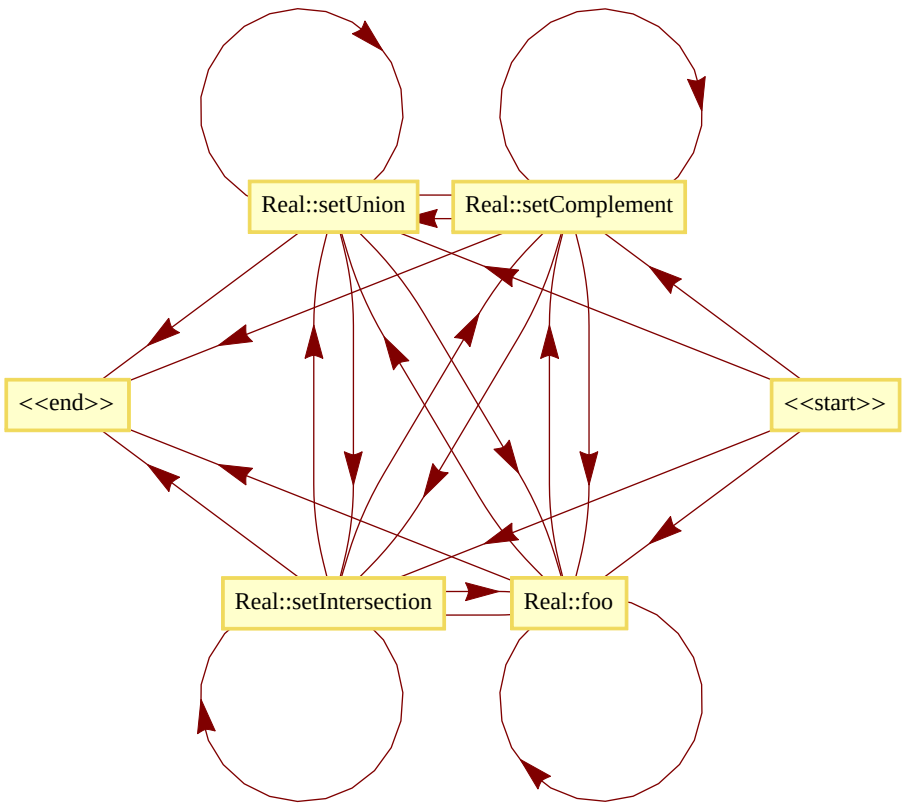


Figure 4.8: Sample CGG for the `setops` problem

A pseudo-method is created for the begin and end nodes. The begin method simply outputs the user's input variables while the end node takes as input the

[1]See the description of backtracking and restarting for a complete clarification of this.

user's desired outputs. Before we begin connecting the graph, we create a map that associates possible inputs and outputs with methods. Recall that an output is anything that the method produces or modifies. The concept type of inputs and outputs is the first cue that two given methods can be connected. For derived classes we include inherited methods in this process (so that the "this" object type can be enumerated for inputs and outputs) and also consider that when a base class is used as a parameter any of its derived classes classes could also be used for that input/output.

After we have created the method input/output map, to connect the CGG, we start with the end node and determine if we can find a path to the start node, creating multiple connections to other method nodes in the process. We perform a breadth-first traversal examining the inputs of the current node for nodes which produce those outputs. We form a connection if at least one input maps to one output but first we do a partial match on the output passed to input as described with the `match()` method. If `match()` returns a non-negative value then a connection is formed. We stop when we have visited every node in the expanding front. If in the process, we have not formed a path from the start node to the end node, we report an error.

4.3.3 Code Generation Process

Each time a new solution is requested by the user, we create an initial state of the input variables and queue the start node for execution. The figure below gives an example of the `mvar` representation of an initial state generated for the sample problem shown in the Summary section.

Note that the state representation includes, most importantly, a "vars" section which maintains the current state of variables (but not their runtime values as the

```
[seq:[[:0, code:= , outs:[a:2, b:2, d:2, f:2]]],
solutionId:41, solutionTag:"f", temp:0,
vars:[Length:[a:[age:0, const:true, enabled:true,
given:true, in:true, inputUses:0, length:0,
out:false, outputUses:0, poly:false,
set:true, takeThis:false, vec:true] .... ]]
```

Figure 4.9: Sample `mvar` representation of the initial state

code generator does not perform an actual execution as it progresses – though this is an approach that I may explore later). The matching information is contained in this section and various values e.g: input/output uses are updated at each node. The current sequence of code generated thus far is contained in the "code" section which also maps the inputs and outputs that were used at each step. A subsequent state after some calls have been performed will look something like the figure below (with some portions of the code emitted for brevity).

```
[lastChoices:[4,6,7,8],
seq:[
0:[:0, code:= ,
outs:[a:2, b:2, d:2, f:2]],

1:[:7, code:{
  Length* tU_0 = b.setIntersection(f);
},
ins:[b:true, f:true], outs:[tU_0:2]],

2:[:6, code:{
  Length* tU_1 = a.setComplement(b);
}, ins:[a:true, b:true], outs:[tU_1:2]],

3:[:4, code:{
  tU_1.foo();
}, ins:[tU_1:true], outs:[tU_1:1]],

4:[:6, code:{
  Length* tU_2 = tU_0.setComplement(tU_1);
...
}, ins:[tU_0:true, tU_3:true], outs:[tU_18:2]], ...

27:[:6, code:{
  Length* tU_19 = tU_14.setComplement(tU_13);
}, ins:[tU_16:true], outs:[tU_16:1]]],

solutionId:30, solutionTag:"U", temp:20,
vars:[Length:[a:[age:0, const:true, enabled:true,
given:true, in:true, inputUses:3, length:0,
out:false, outputUses:0, poly:false, set:true,
takeThis:false, vec:true],...]]]
```

Figure 4.10: Sample `mvar` representation of the state after some calls have been performed

Note here that we can see the call sequence (at the beginning of the `mvar` state) for the path that led to this particular point. We also retain the method id's of other immediate potential branches that could have been made at the last step (as used in the merge and backtracking methods described later).

At each node, we generate a list of the variables that could be potentially applied to a given method parameter. We apply the `match()` to each so that we can create a prioritized map of which variable permutation to perform the call with. A variable permutation is selected stochastically with the probability of that permutation being selected as linearly proportional to the sum of its match priority (over the variable/-parameter matches). We do a simulated run of the method on the chosen variable permutation, including executing postconditions, which modifies the variable state of the selected variables, also creating output variables. We select an output edge to signal uniformly randomly – although some work is underway to allow the system to strengthen connections between methods and thereby modify the probability that the next method will be selected – the problem with this idea is that the proper solution is not only dependent on the code sequence but several features of the state.

It is possible that at a certain branch, a state may fail to produce a variable set that matches because of the more rigorous matching that is done in the full match vs. the partial match that was done in the CGG creation phase when we formed connections between nodes. In this case, we either backtrack or stop and start over again by re-queueing the start node.

At some point we will reach a node which is connected to the end state and that node has some probability of signaling the end node thereby finishing the solution and sending it to the user for evaluation. The probability that the end node is signaled is determined by the `complexity` parameter which is described at the end of this chapter.

When the final node is reached, similar to non-terminal nodes, we choose a variable permutation over which we map temporaries to the final desired outputs. A solution is submitted to a queue for the user to evaluate, but first we perform some normalizations on it. It is likely that several temporaries have been produced that the final desired output(s) do not depend on. We use the input/output maps in the step by step solution data to *normalize* the final solution so that no extraneous calls are included.

When the user is ready to evaluate the solution by calling `next()` on the `CodeGen` instance, the interpretable M code is made available and the host program executes it thereby calling the C++ implementation code. If the code generation process fails, or the implementation throws an exception, then the process is repeated to produce another solution. If code generation succeeds and execution succeeds, either the desired fitness/error has been achieved and we are done or, more likely, the user submits a fitness for the evaluated solution and it is placed in a population of solutions of fixed size (specified by the `population` parameter in the constructor of `CodeGen`). If the population is at its capacity and the solutions's fitness is below that of the lowest fitness solution in the population, the solution is denied entry and is discarded, else it forces the lowest fitness solution to be removed from the population so that the new solution can be inserted.

Because it is possible for a population of solutions to become somewhat homogenous and thereby reach a local optimum, stagnating the code generation process, the `CodeGen` method `reset()` can be called to clear the solution population manually or we can set an automatic periodic reset interval.

4.3.4 Merge

Maintaining a population of solutions allows analysis to be done on high fitness solutions and to use this information to inform the code code generation process about what has worked well in the past. This is a work in progress, therefore currently the primary reason for keeping a population of solutions is to support what I call the "merge" method. Merge works by selecting two high fitness solutions stochastically, merging their state and code, then re-queuing to a selected CGG node.

Doing this right, so that it doesn't cause problems in the code generation process now or later, requires a somewhat complex merging process of the two states. First, we rename the temporaries with a new solution tag to avoid collisions. Then we append the code of the second onto the first. Finally, we re-queue this merged state to one of the solution's last node choices that was available before one of the two solutions signaled the end node. Now, the solution has significantly more state and temporaries to work with from two solutions that have already performed well and can continue on with the code generation process to hopefully produce a better solution.

4.3.5 Static Variables

Meta Concepts features the notion of *static* variables which entail special handling. Note that the name static, while the semantics of such bear some resemblance to the `static` keyword in C++, has an exact meaning specific to Meta Concepts. Input variables can be specified with the `static` attribute. Static variables can be the result of several operations and, the combination of the operations that were used to produce the static variables is collapsed in the sense that the value of a static variable is included directly in the solution of the code, while the calls that went into creating that static variable are removed from the solution. A static appears in a

solution as:

```
Length* length_0 = new Length;
length_0.set(1+1/29);
```

Statics effectively allow the creation and evolution of constants which are often a vital part of a solution, especially in mathematically-oriented problems. The code generator handles statics as follows: a variable is static and can be collapsed if all the variables that went into producing it were also statics. However, in general, statics can also interact with non-static variables.

4.3.6 Control Parameters

Various parameters have been identified as being important in the code generation process but the proper settings of their values varies from problem to problem. Currently, the code generation process can be controlled by the user with the following parameters:

- `mergeRate` μ - when a solution's fitness is submitted, a uniformly random variable x in the range [0..1] is generated such that if $x < \mu$, then a merge will be performed rather than generating a new solution from scratch.
- `selectionPressure` ψ - in the merge process, two solutions are selected according to an exponential distribution which favors fitter solutions. ψ controls the tendency of this selection process to favor fitter solutions with lower ψ values tending to make this selection process more uniform.
- `complexity` c - related to p_e, the probability that the end node with be signaled. $p_e = 1/c$ such that the expected value of the number of terminal nodes which

will be reached before the end node is signaled is equal to `complexity`. Thus complexity can be adjusted to effectively control the length of the generated solutions.

- `minComplexity` - require the solution length to be at least `minComplexity` before allowing the end node to be signaled.

- `maxComplexity` - when solution length reaches `maxComplexity` always signal the end node from terminal nodes.

- `unusedBias` u_b- higher values favor selecting unused variables in the variable permutation as determined by: $m^* = \frac{1}{1/u_b + inputUses}$ and $m_{final} = m \cdot m^* \cdot weight$

- `fill` - true to wait until the population reaches its capacity before beginning to apply the merge operation.

- `restartRate` - the probability multiplier that instead of signaling the next node in the CGG, one of the nodes that comprises the starting method set will be signaled instead. The ability to restart was an important feature that was added later on in the development of Meta Concepts because we can easily reach limited paths through the CGG which can severely limit the possibilities of the generated code.

- `backtrack` - true/false. Similar to restarting, backtracking allows us to back up from the current path one level upon reaching a dead end in the CGG traversal. A dead end is caused by reaching a node in which we cannot map variables to method parameters.

4.4 Meta Concepts GUI

As described in the Motivation section, Meta Concepts can be run in networked mode. This allows the Meta Concepts GUI, a Mac OS X application written using Objective C/Cocoa, to connect to a code generation run in progress, either on a local machine or on a remote machine using TCP/IP. This is particularly useful when we are performing long-running code generation problems whereby we can connect to a remote machine which is hosting the run in order to monitor its progress or to control various aspects of the run. The Meta Concepts GUI currently allows for:

- visualizing the progress of the run via a plot of its progressive best, average, and worst fitness.
- the ability to view the code for solutions of different rank, e.g: view the code and fitness of the solution for the current best solution, or the tenth best solution, etc.
- selectively promote certain "interesting" solutions by giving them higher priority which increases the probability that they will be selected for further modification and merging.
- view a simplified representation of the CGG.
- dynamically alter certain code generation parameters, e.g: selection pressure, merge rate, etc.
- manually reset the code generator thereby clearing the population of solutions.

It is also possible to run the code generator in verbose mode which outputs to the console various statistics about the run and the last generated solution that was accepted into the population.

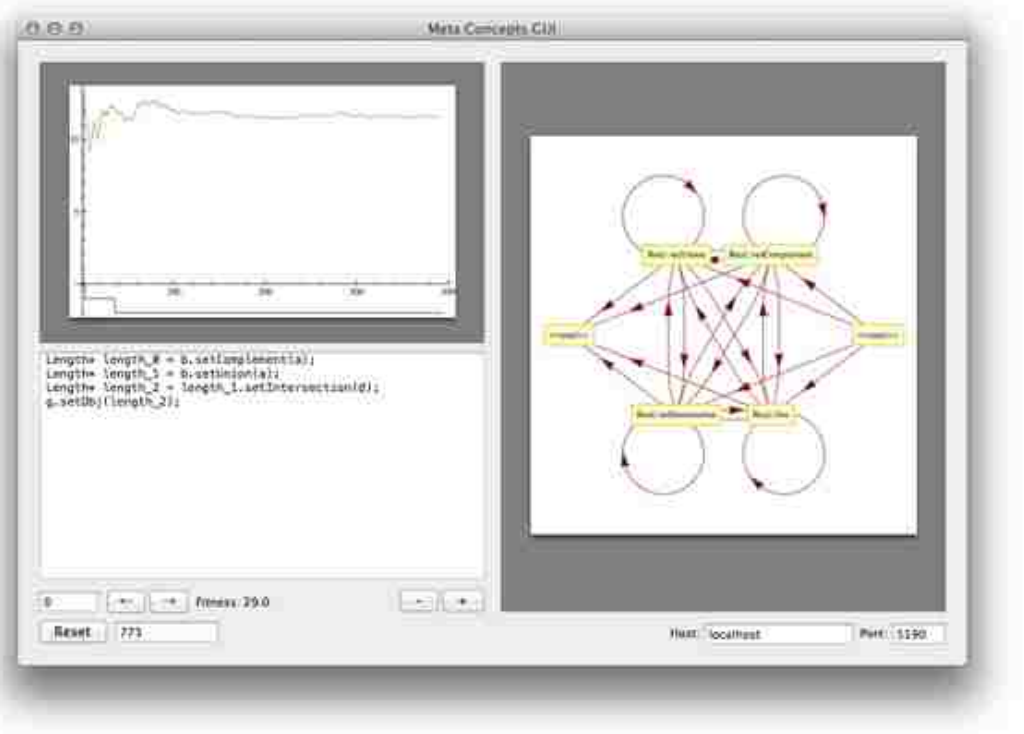


Figure 4.11: The Meta Concepts GUI attached to a `setops` run in progress.

Chapter 5

Results

This chapter presents the results of applying Meta Concepts to three key problems which represent a good mix of the different types of problems that Meta Concepts was designed to solve. The first is the `setops` test case/problem as has been partially described so far. `setops` is a rather simple problem which solves quickly but is sufficiently complex enough to test the effectiveness of the system and is thereby a useful baseline. The second problem to be presented, is significantly more challenging, involving a symbolic regression that derives a series of general equations which describes the area of a regular n-sided polygon as a function of n and the length of its sides l. The third problem attempted, is especially challenging, applying Meta Concepts towards game-playing in which we evolve code that makes a decision on how to make a competitive next move in the board game Mancala.

5.1 Evaluation Approach

In developing, testing, and evaluating the effectiveness of Meta Concepts, there are several important metrics that could potentially be utilized. Perhaps the most im-

portant among them is the number of iterations (i.e: number of solutions generated and evaluated) before reaching an acceptable final solution for a given problem. In this section, we perform several runs of each problem and present the associated runtime metrics. In some cases we were able to perform several runs and present the mean and standard deviation over these trials. In this way we can test the effects of various features by adjusting the parameters or disabling certain functionality in order to evaluate their effectiveness and to evaluate different potential approaches, e.g: how does a certain problem perform with and without merge enabled?

5.2 Metrics

The following list summarizes some of the important computed metrics we utilize in this chapter:

- round - the current round number – a new round is started each time we reset the code generator (i.e: clear the population of solutions), as a result of resetting the code generator all of the metrics that follow are also reset for that round.

- iterations - the number of iterations that have been performed since the beginning of the round. The iteration count is incremented when a successfully generated solution has been presented to the host program for evaluation.

- run time - the total time in seconds that has elapsed for this run (over all rounds).

- round time - the total time in seconds that has elapsed since the start of the round.

- population size - the total number of solutions currently in the population – this will always be less than or equal to the code generator's population size parameter.
- best fitness - the fitness of the highest ranking solution in the population.
- worst fitness - the fitness of the lowest ranking solution in the population.
- average fitness - the mean fitness of all solutions in the population.
- average temps - the mean number of temporaries used over all solutions in the population.
- average normalized size - the mean size of solutions in the population after normalization. Recall that normalization involves removing extraneous called that the final desired output(s) do not depend on.
- average original size - the mean size of solutions in the population before normalization.
- temp utilization - measures the utilization of temporaries by dividing the avg. normalized size by the avg. number of temps.
- average reduction - measures the mean solution size reduction by dividing the avg. original size by the avg. normalized size.
- misses - measures the number of times the code generator reached a dead end and was forced to start over, i.e: it failed to match available variables with call parameters.
- hit rate - related to misses, measures the number of solutions that did not result in a miss divided by the total number of solutions attempted.

- total accepted - measures the total number of solutions that have been accepted into the population since the beginning of the round. There are two possible reasons for a solution not being accepted: 1) it is a duplicate of another solution in the current population. 2) its fitness was below the worst fitness in the population. The total denied metric measures the total number of solutions which were not accepted for these reasons. Note that the accepted and denied counters do not begin being incremented until the population is full if the `fill` parameter is enabled.

- total errors - measures the number of solutions generated which resulted in a runtime exception. Concept classes are free to throw exceptions for various reasons, e.g: divide by 0, invalid index, a negative length, etc.

- error rate - computes the number of errors over the total number of solutions attempted.

- duplicates - measures the number of solutions which were attempted as entered into the population but already existed in the population.

- duplication rate - measures the number of duplicates divided by total number of solutions attempted.

5.3 Hardware and Software Setup

The development and testing of Meta Concepts and some of the shorter runs were performed on a quad-core 2.7 GHz Ivy Bridge 2012 MacBook Pro running Mac OS 10.8. All C++/MML++ code, including the Meta Framework, was compiled using the MML++ compiler which is kept up to date with the development trunk of the Clang compiler. The long-running problems were conducted on a dedicated 2012

Mac Mini with a quad-core 2.6 GHz Ivy Bridge processor whereby 8 independent processes were run in parallel.

5.4 Setops Problem

In the `setops` test case/problem, we perform a simple type of symbolic regression whose minimal hand-written solution requires the proper chaining of 4 calls involving 4 input variables and 1 desired output. Setops is a good baseline because it is simple enough that it runs rather quickly, and thus we can perform several trials to compute the mean and standard deviation of metrics while on the other hand it is complex enough to evaluate the effectiveness of several features used by the system. We present the following sets as input:

$$A = \{1, 2, 3\}; B = \{4, 5, 6\}; F = \{6\}; D = \{1, 2, 5, 6, 7, 8\}; \tag{5.1}$$

where we seek to create an output set:

$$G = \{1, 2, 5\} \tag{5.2}$$

5.4.1 Atomics

The ontology contains the following set of atomics which are selected by the code generation system to form the CGG. A complete listing of these can be found in the Ontology Headers section of the Appendix under `Real.h`. In summary they are:

- `setUnion()` - set union

- `setIntersection()` - set intersection
- `setComplement()` - set complement
- `foo()` - a no-op for testing purposes

5.4.2 Fitness

A solution's fitness is computed as follows: for each element g_i in the target set G, reward 10 fitness points if $g_i \in \{1, 2, 5\}$ else subtract 1 fitness point.

5.4.3 Effectiveness of Merge

To evaluate the effectiveness of merge for this problem, 100 trials were performed with a merge rate of 0.8 (run 1A) and 100 trials with merge rate of 0 (run 1B), keeping all other parameters the same. The results for these runs are reported below.

Run 1A

Run 1A was conducted with the following parameters:

```
trials: 100
fill: false
population: 1000
selectionPressure: 0.5
unusedBias: 5.0
mergeRate: 0.8
restartRate: 1.0
minComplexity: 0
maxComplexity: 9999999
complexity: 10
backtrack: true
resetInterval: 9999999
desiredFitness: 1.79769e308
```

Table 5.1: Run 1A parameters

A graphical representation of the CGG was given in Code Generation section of the Approach chapter. The CGG in text form and related stats are:

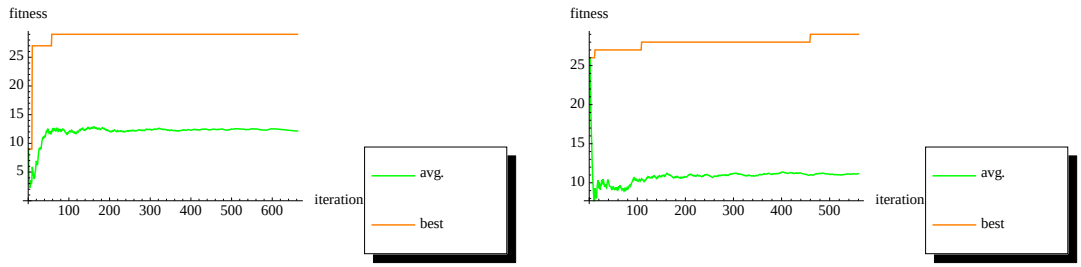
```
Real::setComplement => Real::setIntersection
Real::setIntersection => Real::setIntersection
Real::foo => Real::setUnion
Real::setComplement => Real::setUnion
Real::setIntersection => Real::setUnion
Real::setUnion => Real::setUnion
<<start>> => Real::setUnion
Real::setUnion => Real::setIntersection
<<start>> => Real::setIntersection
Real::setIntersection => Real::setComplement
Real::setUnion => Real::setComplement
<<start>> => Real::setComplement
Real::setComplement => Real::foo
Real::setIntersection => Real::foo
Real::setUnion => Real::foo
<<start>> => Real::foo
Real::foo => <<end>>
Real::setComplement => <<end>>
Real::setIntersection => <<end>>
Real::setUnion => <<end>>

total methods: 39
total nodes: 6
edge upper bound: 24
edges used: 24
unused edges: 0
edge usage: 1
```

Table 5.2: Run 1A CGG data

The output metrics are:

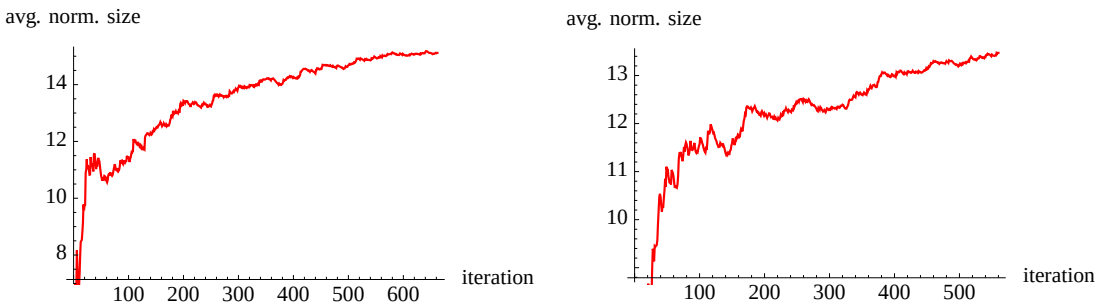
metric	**mean**	**std. deviation**
iterations	1039.93	980.822
round time	10.1488	9.99823
best fitness	28.77	0.617178
worst fitness	0.85	7.11433
avg. fitness	14.8232	4.93185
avg. temps	9.97753	1.78807
avg. norm size	14.7971	2.48766
avg. orig. size	31.5122	5.72639
avg. reduction	2.5879	0.187069
misses	0	0
hit rate	1	0
accepted	793.59	575.523
denied	246.34	458.106
errors	2.13	5.05656
error rate	0.00261239	0.0066635
duplicates	108.16	120.711
duplication rate	0.0820795	0.0359945

Table 5.3: Results for run 1A



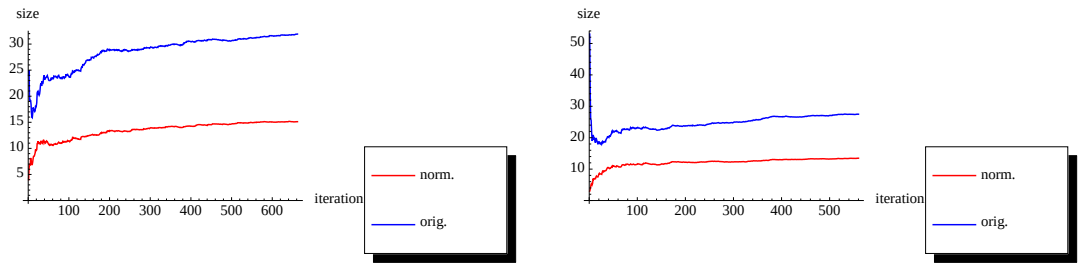
(a) trial X (b) trial Y

Figure 5.1: Sample progressive best and avg. fitness plots from run 1A



(a) trial X (b) trial Y

Figure 5.2: Sample progressive avg. norm. size for the fitness runs plotted above



(a) trial X (b) trial Y

Figure 5.3: Sample progressive avg. solution norm. size and orig. size for the fitness runs plotted above

Note that in the plots depicted above, and with similar plots shown later, the x-axis counting iterations which we will subsequently refer to as *population iterations* corresponds to each time a new solution was admitted into the population – it is different than the overall number of iterations (whose iteration count is also incremented in the case of denial or error) as reported in the previous table which we will continue to simply refer to as *iterations*.

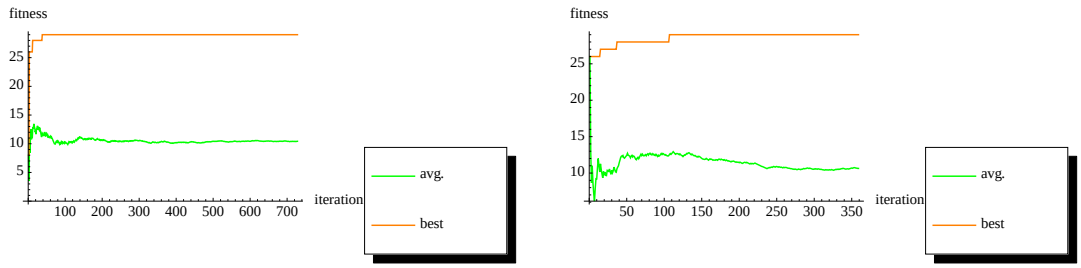
Run 1B

Run 1B was conducted with same parameters as run A, with the only modification being that we changed `mergeRate` = 0.80 to `mergeRate` = 0.0.

The CGG remains unchanged. The output metrics for run 1B are:

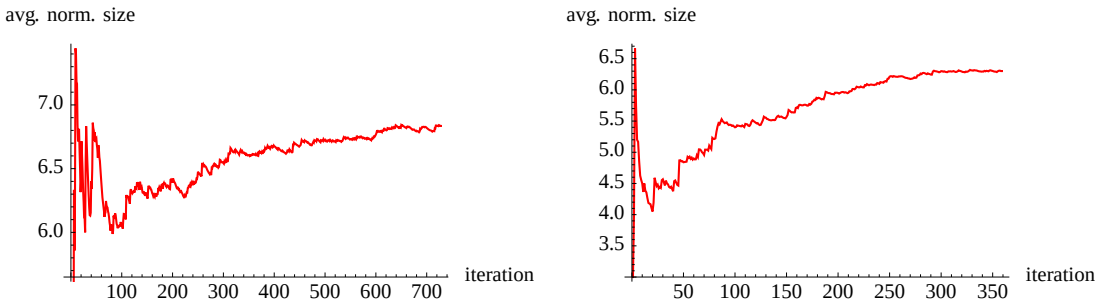
metric	**mean**	**std. deviation**
iterations	1982.27	1664.49
round time	10.9301	9.1798
best fitness	28.86	0.402517
worst fitness	0.3	4.52267
avg. fitness	13.1485	4.24026
avg. temps	3.90992	0.416299
avg. norm size	6.72959	0.485083
avg. orig. size	16.5913	1.48711
avg. reduction	2.53428	0.129423
misses	0	0
hit rate	1	0
accepted	899.18	561.194
denied	1083.09	1137.42
errors	0	0
error rate	0	0
duplicates	845.77	685.423
duplication rate	0.391098	0.0899896

Table 5.4: Results form run 1B



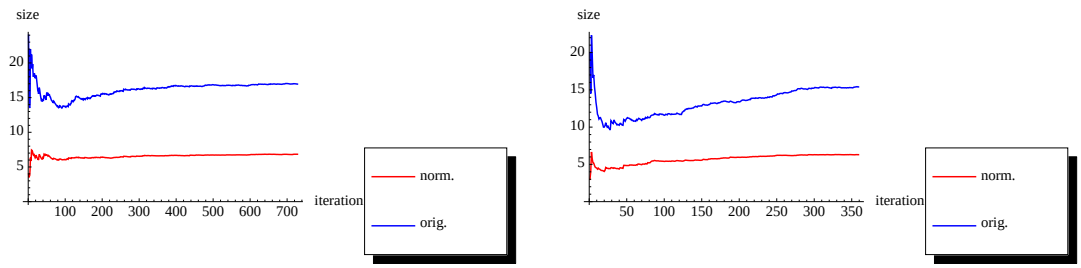
(a) trial X (b) trial Y

Figure 5.4: Sample progressive best and avg. fitness plots from run 1B



(a) trial X (b) trial Y

Figure 5.5: Sample progressive avg. norm. size for the fitness runs plotted above



(a) trial X (b) trial Y

Figure 5.6: Sample progressive avg. solution norm. size and orig. size for the fitness runs plotted above

Final Solutions

Here we present a sample of two final solutions that solve this problem.

```
Length* length_0 = b.setIntersection(f);
Length* length_1 = b.setUnion(a);
Length* length_2 = length_1.setComplement(d);
Length* length_3 = length_0.setIntersection(length_1);
Length* length_4 = length_2.setUnion(f);
Length* length_5 = length_1.setComplement(length_4);
length_5.foo();
Length* length_6 = length_5.setComplement(length_3);
Length* length_7 = length_6.setUnion(length_3);
Length* length_8 = length_7.setComplement(f);
g.setObj(length_8);
```

Table 5.5: Solution 1A

```
Length* length_0 = b.setIntersection(d);
Length* length_1 = length_0.setComplement(f);
length_1.foo();
Length* length_2 = a.setUnion(length_1);
Length* length_3 = length_2.setIntersection(d);
g.setObj(length_3);
```

Table 5.6: Solution 1B

5.5 Area of a Regular n-sided Polygon Problem

In this problem, which we subsequently refer to as `area`, we perform a significantly more complex type of symbolic regression in which we derive equations which relate

the area of regular (all side lengths are the same) n-sided polygon to n and the length l of its sides. The following equations describe this relationship [10]:

$$\varphi = \frac{2\pi}{n} \text{ (center angle)} \tag{5.3}$$

$$l = 2r \sin \frac{\varphi}{2} \text{ (length of sides)} \tag{5.4}$$

$$A = \frac{1}{2} n r^2 \sin \varphi \text{ (area)} \tag{5.5}$$

5.5.1 Atomics

The CGG picks up on a number of methods which are used in the final solution which are summarized as follows. A complete listing of these can be found in the Ontology Headers section of the appendices under the headers for `Real.h`, `Angle.h`, `Length.h`, and `Area.h`.

- A real r can be multiplied by a count to create a real of polymorphic type r. Similarly, division can be applied.
- A real r can be divided by a r ratio to produce a real of polymorphic type r.
- An angle can have the sin() function applied to it to create a length ratio.
- A length can be multiplied by another length to create an area.
- A length ratio can multiplied by an area to create an area.

Inputs

- `Count c` = 2
- `Count n` = # of sides of the polygon
- `Length l` = length of each side
- `Angle t` = $\frac{2\pi}{n}$
- `Angle t2` = $\frac{\pi}{n}$

Outputs

- `Area areaOut`

Evaluation and Fitness

To reduce the adverse effects of awarding high fitness to a solution which by chance solves the problem for any one input set but does not solve the problem overall and also to ensure that we are generating a correct formula for a polygon of arbitrary n sides, the evaluation stage consists of generating and testing polygons of sides $n = [3..10]$ for 10 instances each, given each instance a randomized size for l. The fitness is evaluated using a modified version of the the RMS method given these inputs against the known output area produced from the source equations listed in the opening of this section.

Output Metrics

Because this is a long-running problem which takes roughly on the order of an hour to solve with my current Meta Concepts implementation and hardware setup, it was

difficult to run it for a large number of trials as was done for `setops`. Therefore, we present the run metrics for eight individual runs which successfully solved this problem. Two runs are listed here and the rest can be found in the appendices.

Run 2A parameters, CGG, and metrics:

```
trials: 1
fill: true
population: 3000
selectionPressure: 0.5
mergeRate: 0.8
minComplexity: 0
maxComplexity: 9999999
complexity: 30
backtrack: true
resetInterval: 50000
desiredFitness: 1.79769e308

Area::mulLengthRatio => Area::mulLengthRatio
<<start>> => Length::mulToArea
<<start>> => Real::divCount
Real::divCount => Real::divCount
<<start>> => Real::divRatio
Real::divCount => Real::divRatio
Real::divRatio => Real::divRatio
<<start>> => Real::mulCount
Real::divCount => Real::mulCount
Real::divRatio => Real::mulCount
Real::mulCount => Real::mulCount
```

```
Area::mulLengthRatio => Real::mulCount
Length::mulToArea => Real::mulCount
<<start>> => Angle::sin
Real::divCount => Angle::sin
Real::divRatio => Angle::sin
Real::mulCount => Angle::sin
Angle::sin => Real::mulCount
Real::mulCount => Real::divRatio
Area::mulLengthRatio => Real::divRatio
Length::mulToArea => Real::divRatio
Angle::sin => Real::divRatio
Real::divRatio => Real::divCount
Real::mulCount => Real::divCount
Area::mulLengthRatio => Real::divCount
Length::mulToArea => Real::divCount
Angle::sin => Real::divCount
Real::divCount => Length::mulToArea
Real::divRatio => Length::mulToArea
Real::mulCount => Length::mulToArea
Length::mulToArea => Area::mulLengthRatio
Real::divCount => Area::mulLengthRatio
Real::divRatio => Area::mulLengthRatio
Real::mulCount => Area::mulLengthRatio
Angle::sin => Area::mulLengthRatio
Area::mulLengthRatio => <<end>>
Length::mulToArea => <<end>>
Real::divCount => <<end>>
Real::divRatio => <<end>>
```

```
Real::mulCount => <<end>>

total methods: 21
total nodes: 8
edge upper bound: 48
edges used: 40
unused edges: 8
edge usage: 0.833333

----- round:            0
----- iterations:       7331
----- run time:         954.559
----- round time:       954.559
----- population size:  3000
----- best fitness:     1.3438e-08
----- worst fitness:    2.5479e-13
----- avg. fitness:     1.8731e-10
----- avg. temps:       7.19933
----- avg. norm. size:  9.19933
----- avg. orig. size:  45.814
----- temp util:        1.2778
----- avg. reduction:   5.03673
----- total misses:     4733
----- hit rate:         0.667159
----- total accepted:   581
----- total denied:     390
----- total errors:     2155
----- error rate:       0.151547
```

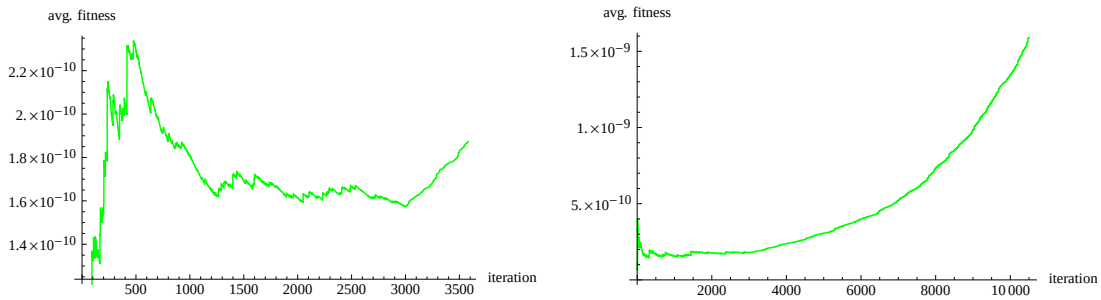
```
----- total duplicates: 3696
----- duplication rate: 0.50416
```

The parameters and CGG for run 2B are the same as in run 2A; the observed metrics are:

```
----- round:            0
----- iterations:       46794
----- run time:         5165.97
----- round time:       5165.97
----- population size:  3000
----- best fitness:     6.43295e-08
----- worst fitness:    3.69305e-10
----- avg. fitness:     1.59007e-09
----- avg. temps:       10.2357
----- avg. norm. size:  12.2357
----- avg. orig. size:  50.5613
----- temp util:        1.1954
----- avg. reduction:   4.3407
----- total misses:     18371
----- hit rate:         0.7469
----- total accepted:   7504
----- total denied:     32891
----- total errors:     7418
----- error rate:       0.102199
----- total duplicates: 13477
----- duplication rate: 0.288007
```

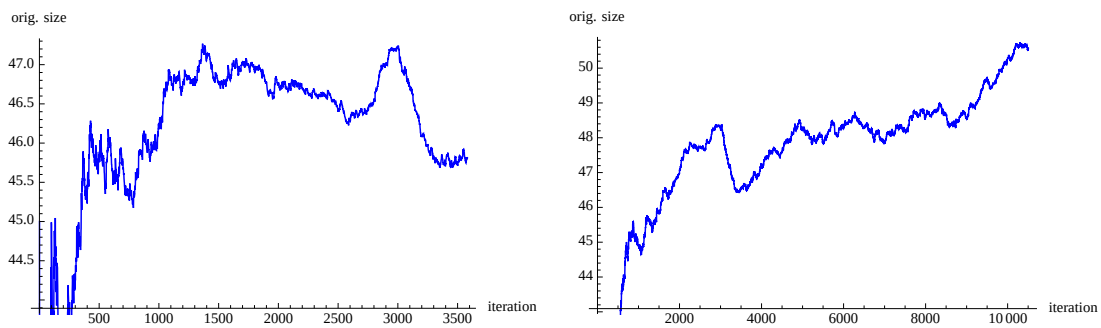
Table 5.7: Run 2B metrics

Metrics



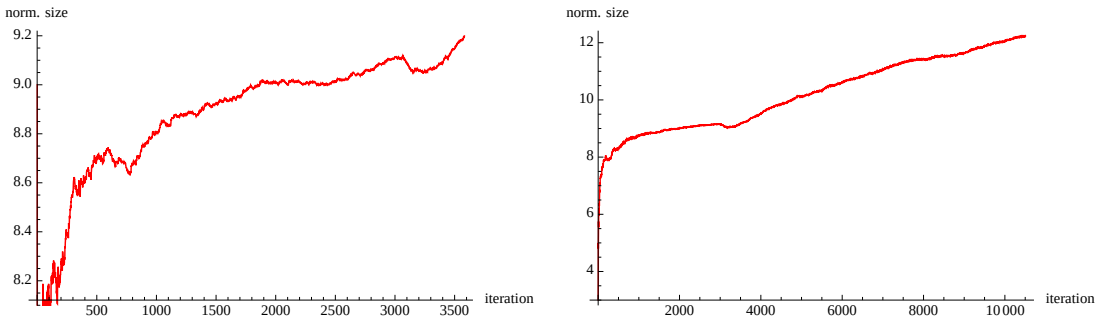
(a) run 2A (b) run 2B

Figure 5.7: Progressive average fitness for run 2A and 2B



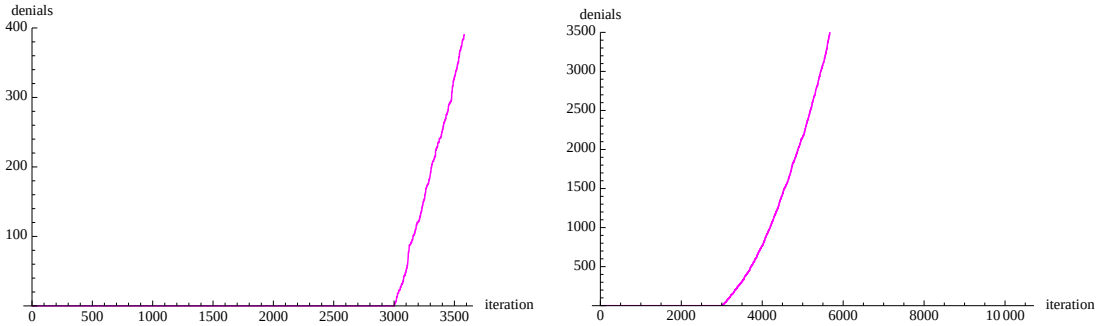
(a) run 2A (b) run 2B

Figure 5.8: Progressive average original size for run 2A and 2B



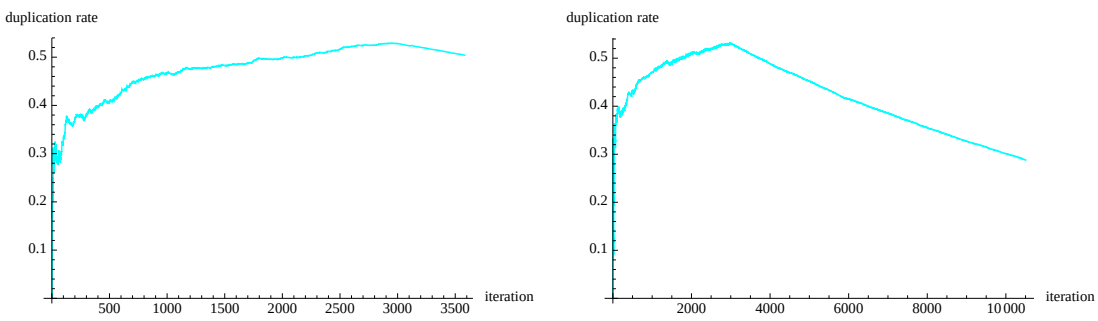
(a) run 2A (b) run 2B

Figure 5.9: Progressive average normalized size for run 2A and 2B



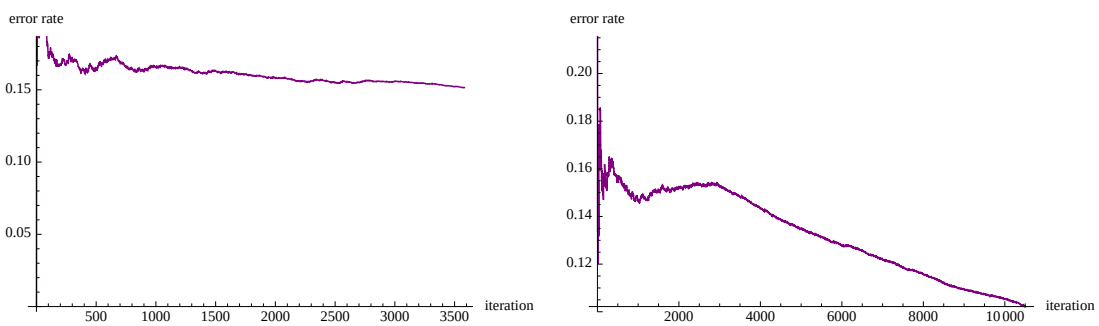
(a) run 2A (b) run 2B

Figure 5.10: Progressive total number of denials for run 2A and 2B



(a) run 2A (b) run 2B

Figure 5.11: Progressive duplication rate run 2A and 2B



(a) run 2A (b) run 2B

Figure 5.12: Progressive error rate for run 2A and 2B

Final Solutions

We present the final solutions for run 2A and run 2B. The solutions for the remaining six runs, as well as additional output metrics and plots are listed in the appendices.

```
Length* length_0 = t.sin();
Length* length_1 = l.divCount(c);
Length* length_2 = length_1.divRatio(length_0);
Length* length_3 = t2.sin();
Length* length_4 = length_2.divRatio(length_3);
Area* area_5 = length_4.mulToArea(length_4);
Length* length_6 = t.sin();
Length* length_7 = length_6.divCount(c);
Length* length_8 = length_7.mulCount(c);
Area* area_9 = area_5.mulLengthRatio(length_8);
Length* length_10 = length_7.mulCount(c);
Length* length_11 = length_8.mulCount(n);
Area* area_12 = area_9.mulLengthRatio(length_11);
Area* area_13 = area_12.mulLengthRatio(length_10);
Area* area_14 = area_13.divCount(c);
Area* area_15 = area_14.mulCount(n);
Area* area_16 = area_15.divCount(n);
areaOut.setObj(area_16);
```

Table 5.8: Solution 2A

```
Length* length_0 = l.mulCount(c);
Length* length_1 = t2.sin();
Length* length_2 = l.divRatio(length_1);
Length* length_3 = t.sin();
Length* length_4 = length_2.divRatio(length_3);
Length* length_5 = length_4.divRatio(length_1);
Area* area_6 = length_0.mulToArea(length_5);
Area* area_7 = area_6.divCount(c);
Length* length_8 = length_3.divCount(c);
Area* area_9 = area_7.divCount(c);
Area* area_10 = area_9.mulCount(n);
Area* area_11 = area_10.mulLengthRatio(length_8);
Area* area_12 = area_11.mulLengthRatio(length_8);
areaOut.setObj(area_12);
```

Table 5.9: Solution 2B

5.6 Mancala Problem

For the final problem, `mancala`, moving away from symbolic regression, we attempt to evolve a strategy for playing the game Mancala, taking advantage of the system's ability to interface with complex implementation code. This is a good choice for Meta Concepts, because the rules/mechanics of the game can be coded as a concept/class then the next choice of move is distilled down to choosing an integer [1..6] which corresponds to our move decision whereby we first do some analysis on the current state of the board in order to (hopefully) make an intelligent/competitive next move. The Mancala game board looks like:

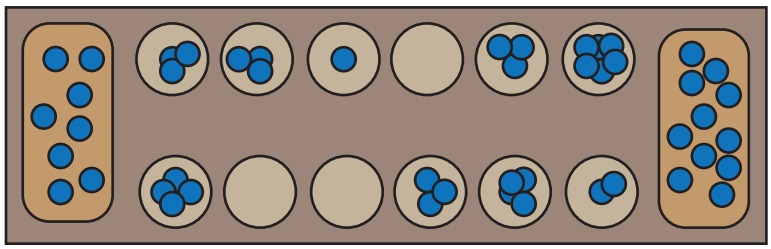


Figure 5.13: Sample Mancala board

The rules of the game are summarized as follows [11]:

- The Mancala board is made up of two rows of six slots each.
- The initial configuration is created by placing four pieces placed in each of the 12 slots.
- Each player has a "store" to the right side of the Mancala board.

- The game begins with one player picking up all of the pieces in any one of the slots on his side.
- Moving counter-clockwise, the player deposits one of the pieces in each slot until the pieces run out.
- If you run into your own store, deposit one piece in it. If you run into your opponent's store, skip it.
- If the last piece you drop is in your own store, you get a free turn.
- If the last piece you drop is in an empty slot on your side, you capture that piece and any pieces in the hole directly opposite. Always place all captured pieces in your store.
- The game ends when all six slots on one side of the Mancala board are empty.
- The player who still has pieces on his side of the board when the game ends captures all of those pieces.
- Count all the pieces in each store. The winner is the player with the most pieces.

5.6.1 Implementation and Atomics

To formulate this problem as a Meta Concepts problem, the `Mancala` concept/class was created which makes the following methods available to the code generator [1]:

```
/*[
   description: "return the number of pieces at slot i"
```

[1] Our slots are numbered 1 through 6; our store is slot 0; and the opponent's slots are 7 through 12.

```
]*/
Count* numPieces(const Index* i) const;

/*[
  description: "return the slot index after making a ↩
     number of hops c starting at slot i"
]*/
Index* slotOffset(const Index* i, const Count* c) ↩
   const;

/*[
  description: "return the number of hops required to ↩
     reach slot a from slot b"
]*/
Count* slotDiff(const Index* a, const Index* b) const;

/*[
  description: "return true (1.0) if the specified ↩
     slot is mine"
]*/
Truth* isMine(const Index* i) const;

/*[
  description: "return the AND of a and b = a*b"
]*/
Truth* truthAnd(const Truth* a, const Truth* b) const;

/*[
```

```
  description: "return the OR of a and b = max(a,b)"
]*/
Truth* truthOr(const Truth* a, const Truth* b) const;

/*[
  description: "return the NOT of a = 1.0 - a"
]*/
Truth* truthNot(const Truth* t) const;

/*[
  description: "return 1.0 if c and i are equal else ↩
     0.0"
]*/
Truth* countIndexEqual(const Count* c, const Index* i)↩
    const;

/*[
  description: "return 1.0 if c > i else 0.0"
]*/
Truth* countIndexGreater(const Count* c, const Index* ↩
   i) const;

/*[
 "return 1.0 if c < i else 0.0"
]*/
Truth* countIndexLesser(const Count* c, const Index* i↩
   ) const;
```

```
/*[
  description: "return 1.0 if a and b are equal else ↩
     0.0"
]*/
Truth* countEqual(const Count* a, const Count* b) ↩
   const;

/*[
  description: "return 1.0 if a > b else 0.0"
]*/
Truth* countGreater(const Count* a, const Count* b) ↩
   const;

/*[
  description: "return 1.0 if a < b else 0.0"
]*/
Truth* countLesser(const Count* a, const Count* b) ↩
   const;

/*[
  description: "return a + b"
]*/
Count* countAdd(const Count* a, const Count* b) const;

/*[
  description: "return a - b"
]*/
Count* countSub(const Count* a, const Count* b) const;
```

These methods throw exceptions on various errors cases, e.g: specifying an invalid slot number.

5.6.2 Problem Formulation and Fitness

To decide which move to make, we are effectively generating a sequence of calls which results in a `Truth` value `truthOut`. We perform this sequence of calls successively on each of the six slots: $slot = [1..6]$ and pick as our move the slot which results in the highest `truthOut` value. After we have chosen our move, the `Mancala` concept internally executes the mechanics involved with our choice. A decision was made to make the opponent always perform a random move by choosing a uniformly random slot [1..6] and we always get to make our move first. To reduce the "luck" factor from our results, 50 matches are performed to evaluate each solution and the fitness is the mean of the number of pieces in our store that we have achieved at the end of each match. As a useful baseline, it was determined experimentally that when we make the first move and play randomly against the opponent also playing randomly, that the mean fitness achieved is approximately 32.

5.6.3 Inputs and Outputs

The inputs are:

- `Mancala mancala` - the Mancala game board as a `const`.
- `Index slot` - the current slot we are testing with `weight = 100` to ensure that this variable is highly utilized in solutions.
- `Index index0 = 0` through `index12 = 12` - these correspond to the slot indices.

- `Count count0 = 0` and `count1 = 1` - these counts can be used to arithmetically build up higher count numbers.

The desired output is:

- `Truth truthOut` - we test each of the six slots on our side with the generated solution and pick the slot which corresponds to the highest `truthOut` value.

5.6.4 Output Metrics

8 parallel runs were conducted which ran continuously for several hours. We present the results of the top two runs, run 3A and 3B.

The CGG is rather long and is listed in in the appendices. The parameters for run 3A and 3B are listed below.

```
trials: 1
fill: true
population: 3000
selectionPressure: 0.5
unusedBias: 1.0
mergeRate: 0.8
restartRate: 1.0
minComplexity: 0
maxComplexity: 9999999
complexity: 40
backtrack: true
resetInterval: 200000
desiredFitness: 1.79769e308
```

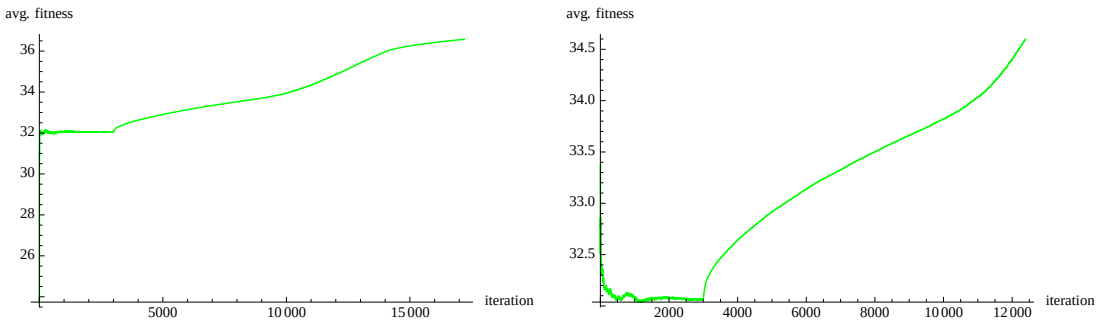
Table 5.10: Run 3A and 3B parameters

```
----- round:            0
----- iterations:       147354
----- run time:         89973.9
----- round time:       89973.9
----- population size:  3000
----- best fitness:     37.8
----- worst fitness:    36.24
----- avg. fitness:     36.5765
----- avg. temps:       13.016
----- avg. norm. size:  15.016
----- avg. orig. size:  55.957
----- temp util:        1.15366
----- avg. reduction:   4.07225
----- total misses:     7611
----- hit rate:         0.954686
----- total accepted:   14209
----- total denied:     129353
----- total errors:     20609
----- error rate:       0.1227
----- total duplicates: 13473
----- duplication rate: 0.0914329
```

Table 5.11: Run 3A output metrics

```
----- round:            0
----- iterations:       63391
----- run time:         50503
----- round time:       50503
----- population size:  3000
----- best fitness:     37.8
----- worst fitness:    33.8
----- avg. fitness:     34.5976
----- avg. temps:       7.33633
----- avg. norm. size:  9.33633
----- avg. orig. size:  59.1073
----- temp util:        1.27262
----- avg. reduction:   8.29488
----- total misses:     4528
----- hit rate:         0.936982
----- total accepted:   9388
----- total denied:     50222
----- total errors:     8462
----- error rate:       0.117768
----- total duplicates: 7658
----- duplication rate: 0.120806
```

Table 5.12: Run 3B output metrics



(a) run 3A (b) run 3B

Figure 5.14: Progressive average fitness for run 3A and 3B

Final Solutions

We present the final solutions for run 3A and run 3B.

```
Index* index_0 = mancala.slotOffset(index12, index12);
Count* count_1 = mancala.numPieces(slot);
Count* count_2 = mancala.slotDiff(slot, index_0);
Truth* truth_3 = mancala.countEqual(count_2, count_1);
Truth* truth_4 = mancala.truthNot(truth_3);
Truth* truth_5 = mancala.truthNot(truth_4);
Truth* truth_6 = mancala.truthAnd(truth_5, truth_5);
Truth* truth_7 = mancala.truthAnd(truth_6, truth_6);
Truth* truth_8 = mancala.truthOr(truth_7, truth_7);
truthOut.setObj(truth_8);
```

Table 5.13: Solution 3A

```
Count* count_0 = mancala.numPieces(slot);
Truth* truth_1 = mancala.countIndexEqual(count_0, slot);
Truth* truth_2 = mancala.truthAnd(truth_1, truth_1);
Truth* truth_3 = mancala.truthAnd(truth_2, truth_2);
Truth* truth_4 = mancala.truthAnd(truth_3, truth_3);
truthOut.setObj(truth_4);
```

Table 5.14: Solution 3B

5.7 Discussion

Meta Concepts was applied to solve a simple symbolic regression in the `setops` problem. Once we developed the system well enough to solve `setops` and other simple problems, we moved toward solving a rather complex symbolic regression

with the `area` problem. `area` was chosen because it is somewhat difficult to derive this relationship by hand using direct mathematical reasoning. We were able to define a set of rules and constraints that greatly reduce the search space using a combination of concept-typing and metadata attributes. The success of this run is a good indication that Meta Concepts will be able to tackle increasingly complex symbolic regression problems in the future as the system is improved.

On the other hand, one of the features that makes these types of symbolic regression problems difficult to solve with Meta Concepts is the often "hit or miss" nature of the fitness measure. In `setops` we were able to define a strong fitness measure that rewards partially correct solutions whereby the merge operation was able to successively build up a final solution. In the case of `area`, without knowing what the final solution looks like or if it even exists, it is significantly more difficult to devise a fitness measure that rewards solutions which are partially correct. How do we know when we are the right track? For example, a solution could be close to being correct, but even if it is only one call away from being ideal, its fitness may be stuck close to 0.

Because `setops` runs quickly, we were able to perform 100 trials each for run 1A and run 1B and judging from the mean and std. deviation of the number of iterations to solution, this gives a strong indication that the merge operation is working well, decreasing the mean number of iterations by about half. Note that the closeness in computational time between run 1A and 1B can be explained by the fact that merge is a computationally expensive operation with the current Meta Concepts implementation and when we do not perform merging, we do not have this extra overhead – but for more challenging problems the extra overhead is well worth the price paid. Based on some preliminary runs that I have completed which I have not presented here, using slightly more complex symbolic regressions, merge can decrease the mean number of iterations required significantly more when a good

fitness measure is used which rewards partially correct solutions.

We also observed that average fitness is significantly higher in run 1A vs. 1B. And in the case of 2A and 2B, that merging tends to increase the average fitness exponentially as the run progresses. This is not surprising as we are recombining already high performing solutions and the temporaries they use to create new solutions, which has a tendency to make the population more homogenous. As we can observe from the plots, this also has the effect of increasing the average and normalized sizes, decreasing the error rate, and also tends to decrease the duplication rate, though this last observation comes as somewhat of a surprise.

As the plots of average and best fitness show, it can be helpful to keep an eye on these metrics as the run progresses. As the average fitness approaches the best fitness, the runs tend to stagnate and make less progress, then we are relying more on mutation vs. merge to chance upon the next change that brings the population to a higher fitness level. If we observe no significant changes to these metrics for some time, this can be a sign that the round has stagnated and we should reset the population. Also note, that when the population is ramping up to full, the average fitness tends to drop until we max out the population and then it begins to climb again.

It is interesting to observe the effects of normalizing the solutions. We can see by the plots that it is common to see an average reduction in size of 2 - 7 times as we remove extraneous calls that the final outputs do not depend on. We can also see the diminished use of longer original solutions as the size of the original solution tends to grow at a higher rate vs. the size of the normalized solutions. Also, as expected, when we observe the mean sizes for run 1A vs. 1B we can see that when merge is enabled, we see much higher average original and normalized solution sizes.

In the plot of denials for run 2A and 2B we observe that after the population

reaches its capacity at 3000, denials grow rapidly as duplicate solutions and solutions with fitness less than the worst population fitness are rejected. Keeping an eye on the number of duplicates and duplicate rate can give an important indication of why exactly this is happening.

The final solutions for run 2A and 2B and those listed in the appendices are insightful. We can see that there are multiple final solutions to the problem which may look entirely different but are equally correct. Looking at the final solutions for 1A and 1B too, we can observe that many of the solutions we achieve contain *introns*. The term intron was coined by the genetic programming community to describe code bloat exhibited by solutions which does not help or hurt the final outcome [9]. Examples of introns include: multiplying a real number by 1, taking the intersection of a set with itself, $a + b - b$, and so on. We are already normalizing the solution to some degree by removing the extraneous calls that the final desired outputs(s) do not depend on, but the task of automatically removing introns poses a more difficult problem because their proper removal depends on the semantics of the calls being made. Introns are not a major cause of concern at this stage of development in Meta Concepts, because they do not harm the effectiveness of the solution and once we have a final solution, in many cases we can remove them from our final solution by hand inspection, however they do make it harder to quickly make sense of a solution initially.

When we examine the additional plots of the metrics for `area`, as listed in the appendix, we can see that behavior of the system follows a predictable pattern, e.g: the curves for average fitness tend to increase at a similar rate across runs as for average normalized size, etc. In many cases, the shapes of the curves are a recurrent signature of the problem and the code generation setup and given code generation parameters. However the curves for different problems look largely different, comparing the average fitness plots of `area` with `mancala`, for instance. Though not

depicted in this thesis, this type of uniformity for a given problem across multiple runs was observed for other problems tackled as well. Note in some of these we see multiple curves due to multiple rounds being performed and resetting the metrics at the end of each round.

Monitoring certain metrics such as the miss rate and error rate can be helpful in debugging problems with the problem setup of ontology. One of the features that was added recently was the ability to have the code generator write a match log and error log which can be useful in diagnosing problems when we observe abnormally high error rates and misses. The match log writes out each attempt to map state variables to call parameters, listing the variables that are used to form a permutation, the overall match priority, when and why we backtracked, and so on. The error log records the exceptions that were thrown during solution execution.

When compared to `area`, the results for `mancala` are less impressive. We were able to improve the fitness from the baseline of 32 to about 38 in the best solution because the generated solutions picked up on the strategy that it should move those pieces at the slot which end up resulting in ending in our store, yielding a free turn. I originally anticipated that the system would be able to achieve a more complex final solution, consisting of logically chaining together combinations of strategies with AND/OR/NOT to also consider choosing the slot which results in a capture, for instance. There are a variety of different strategies that arise as a result of the rules which are required to attain a higher fitness. Further analysis and work is required perhaps to improve both the code generator and the Mancala concept to achieve better results here, but I am optimistic. In the future, it is worth spending time improving `mancala`, because this is an important problem that is prototypical of many agent-based problems of interest.

Some of the problems that were encountered while attempting to tackle `area` and `mancala` provided important insights on how to improve the code generation system.

For example, observing high hit rates and the final solutions, motivated the need for backtracking and solution restart as was found that we cannot rely entirely on the structure of the CGG to determine the code generation path so we need to allow some chance of starting further call paths at the beginning otherwise we severely restrict solution possibilities.

The complexity or challenge of a problem is determined by a number of factors including:

- the number of output variables we are seeking.
- the number of inputs we provide.
- the number of atomics selected by the CGG and how greatly the variables in question are constrained by metadata attributes.
- the minimum number of calls required by a correct solution.

Unfortunately, most of these factors scale the complexity in a highly non-linear fashion. For example, adding one more required call for Meta Concepts to solve a problem which is already performing well can increase the complexity significantly. Therefore, in order to effectively use Meta Concepts to solve a given problem, there are a number of things which should be done to increase the system's chances of success:

- Reduce the number of calls required by providing more or requiring less, i.e: providing more inputs for those values we already know are part of the solution, or limiting the complexity of the final desired outputs in a similar way, e.g: in `area` we provided a hint input: the center angle $\frac{2\pi}{n}$ which enabled us to solve the problem more quickly.

- Narrow the number of atomics by removing/disabling methods which we know or have a hunch are not part of the solution.
- Limit the number of input or output variables so that Meta Concepts only utilizes those variables we need for the solution or assign them appropriate weights.
- Reduce the complexity by reformulating the problem, e.g: as we did for the Mancala problem where we test the solution on each slot number 1 through 6 rather than having solutions return to us the desired slot number of the next move.

Chapter 6

Future Work

Meta Concepts is a new software system which I plan to continue to actively develop in the future. As the system is applied to increasingly complex problems, analysis provides insight into several areas in which the system could be improved upon, including:

- continuing to build up a reusable ontology that encodes higher-level domain-knowledge through attributes and other types of metadata.
- to allow the ontology to learn as it tackles more and more problems, e.g: which methods work well together?
- continuing to improve the Mancala problem so that it can provide highly competitive strategies as it is prototypical of many of the types of agent-based problems that Meta Concepts is concerned with.
- improve the ability to diagnose problems when the system fails to generate good solutions; adding the match log and error log was a good start to this.
- though challenging, research new ways of specifying fitness so that the system

is is better able to reward partial fitness to solutions which are on the right track in order to avoid the "hit or miss" fitness problem.

- specialized functionality for mutation of static constants, instead of relying entirely on the code generator to perform this operation which is slow to produce results.

- enhance the matching system so that it could possibly benefit from CP and CSP solvers.

- continue to develop ways of allowing the user to control the run with the Meta Concepts GUI; the ability to promote select solutions is one of example of this.

- add the ability to match methods and generated code with their mathematical equivalents so that simplification and removal of introns is made possible. This would also have the benefit of making solutions more understandable at a glance.

Chapter 7

Conclusion

This thesis was inspired by earlier work and research that I conducted independently in the area of automatic code generation. I found the idea of computers automatically creating code which solves useful and challenging problems to be most interesting and an incredibly worthy area of effort, despite its inherent difficulty. I later discovered genetic programming as a field which has a similar purpose with much prior interesting and fruitful work which could be learned from. The Meta Framework was designed from the very beginning to support automated code generation as one of its major use cases. This is indeed a very hard problem and throughout the years, I attempted several different approaches but many of them suffered from an inability to scale towards solving increasingly difficult problems. This combined experience and lessons learned have led towards the creation of Meta Concepts, which I believe is a strong step towards my vision of using automated code generation to solve complex problems, and in general, to enable computers to make useful scientific discoveries.

In this project, I developed the basic mechanics of the system by applying it towards simple problems such as `setops`, then once I had this working well, I scaled the system further so that I was able to successfully solve more complex symbolic

regressions, such as `area`. I believe my approach with the `mancala` problem is a good one, but further work is needed to enable it to produce more competitive results. In the process of applying Meta Concepts towards these problems and the accompanied analysis, I gained several important insights on how to expand the capabilities of the system. Some of these are a work in progress while other improvements have been applied already and are working well. As it stands now, Meta Concepts is a powerful code generation system which can be applied to real-world programs written in C++ whereby the final solutions produced can be easily integrated with such programs. A major strength of the system stems from the fact that many problems, in general, can be rephrased as the task of generating a program, making Meta Concepts a highly universally-applicative problem solving tool. Going forward, there remains a significant number of challenges and further work to be done. I look forward to continuing to develop and scale Meta Concepts in the future.

References

- Wikipedia entry for Genetic programming. (3/15/13) http://en.wikipedia.org/wiki/Genetic_programming. [1]
- Wikipedia entry for Genetic algorithms. (3/15/13) http://en.wikipedia.org/wiki/Genetic_algorithm. [2]
- Rina Dechter. (2003) Constraint Processing. Elsevier. [3]
- Farid Ajili, Alexander Bockmayr, et. al. (2004) Constraint and Integer Programming: Towards a Unified Methodology. Springer. [4]
- Krzysztof R. Apt. (2003) Principles of Constraint Programming. Cambridge University Press. [5]
- Christopher Moore, Stephan Mertens. (2011) The Nature of Computation. Oxford University Press. [6]
- Kim Marriott, Peter J Stuckey. (1998) Programming With Constraints. MIT Press. [7]
- Wikipedia entry for Constraint programming. (3/12/13) http://en.wikipedia.org/wiki/Constraint_programming. [8]
- Riccardo Poli, William B. Langdon, Nicholas F. McPhee, John R. Koza. (2008) A Field Guide to Genetic Programming. Published under a Creative Commons

license. [9]

- Eberhard Zeidler. (2004) Oxford Users' Guide to Mathematics. Oxford University Press. [10]
- Mancala Rules from about.com. (4/19/13) http://boardgames.about.com/cs/mancala/ht/play_mancala.htm [11]

Appendices

Appendix A1: Sample Program Setup

```
#include <iostream>

#include <Meta/MLib.h>
#include <Meta/MTime.h>
#include <Meta/MMLGenerator.h>

#include "Length.h"
#include "Ontology.h"
#include "CodeGen.h"

using namespace std;
using namespace Meta;

int main(int argc, char** argv){
  MProgram program(argc, argv);
```

```
// get the ontology singleton
Ontology* ontology = Ontology::get();

// instantiate a code generator with population 10000
CodeGen cg(10000);

// set code generation control parameters
cg.setSelectionPressure(1.0);
cg.setMergeRate(0.5);

// describe the inputs
Length a;
a.setConst(true);
a.setVec(true);
a.setSet(true);

Length b;
b.setConst(true);
b.setVec(true);
b.setSet(true);

Length d;
d.setConst(true);
d.setVec(true);
d.setSet(true);

Length f;
f.setConst(true);
```

```
f.setVec(true);
f.setSet(true);

// describe the outputs
Length g;
g.setVec(true);
g.setSet(true);

// add the variables to the code generator
cg.addInput("a", &a);
cg.addInput("b", &b);
cg.addInput("d", &d);
cg.addInput("f", &f);
cg.addOutput("g", &g);

// begin generating solutions in the background
cg.generate();

size_t i = 1;
for(;;){
  ++i;

  // reset the code generator every 5000 cycles --
  // perhaps we have converged to a local optimum if
  // we haven't found a solution by this time
  if(i % 5000 == 0){
    cg.reset();
  }
```

```
// get the next solution
cg.next();

// inputs
a = [1,2,3];
b = [4,5,6];
f = [6,];
d = [1,2,5,6,7,8];

// run the solution, an error may have occurred
if(!cg.run()){
  continue;
}

const mvar& v = g.val();

// evaluate the fitness
double fitness = 0;
for(size_t i = 0; i < v.size(); ++i){
  if(v[i] == 1 || v[i] == 2 || v[i] == 5){
    fitness += 10;
  }
  else{
    fitness -= 1;
  }
}
```

```
    // desired fitness achieved, print solution and exit
    if(fitness == 30){
      mnode f = cg.getSolution();

      mstr mml = MMLGenerator::toStr(f);

      cout << "------------ iterations: " << i << endl;
      cout << "------------ final solution" << endl;
      cout << mml << endl;
      exit(0);
    }

    // submit the fitness of the solution just evaluated
    cg.finish(fitness);
  }

  return 0;
}
```

Appendix A2: Setops Program Setup

```
#include <iostream>

#include "Length.h"
#include "MCProgram.h"
```

```
using namespace std;
using namespace Meta;

class Program : public MCProgram{
public:
  Program(int argc, char** argv)
    : MCProgram(argc, argv){

  }

  void setup(){
    // describe the inputs
    a.setConst(true);
    a.setVec(true);
    a.setSet(true);

    b.setConst(true);
    b.setVec(true);
    b.setSet(true);

    d.setConst(true);
    d.setVec(true);
    d.setSet(true);

    f.setConst(true);
    f.setVec(true);
    f.setSet(true);
```

```
    // describe the outputs
    g.setVec(true);
    g.setSet(true);

    // add the variables to the code generator
    addInput("a", &a);
    addInput("b", &b);
    addInput("d", &d);
    addInput("f", &f);
    addOutput("g", &g);

    // inputs
    a = [1,2,3];
    b = [4,5,6];
    f = [6,];
    d = [1,2,5,6,7,8];
}

mvar evaluate(){
    if(!run()){
        return undef;
    }

    const mvar& v = g.val();

    // evaluate the fitness
    double fitness = 0;
    for(size_t i = 0; i < v.size(); ++i){
```

```
      if(v[i] == 1 || v[i] == 2 || v[i] == 5){
        fitness += 10;
      }
      else{
        fitness -= 1;
      }
    }

    if(fitness == 30){
      return mvar::inf();
    }

    return fitness;
  }

private:
  Length a;
  Length b;
  Length d;
  Length f;
  Length g;
};

int main(int argc, char** argv){
  Program program(argc, argv);

  program.init();
  program.start();
```

```
  return 0;
}
```

Appendix A3: Area Program Setup

```
#include <iostream>

#include <cmath>

#include <Meta/MLib.h>

#include "Length.h"
#include "Area.h"
#include "Angle.h"
#include "Count.h"
#include "MCProgram.h"

using namespace std;
using namespace Meta;
using namespace Func;

class Program : public MCProgram{
public:
  Program(int argc, char** argv)
    : MCProgram(argc, argv){
```

```
}

void setup(){
  t.setConst(true);
  t2.setConst(true);

  c.setConst(true);
  c = 2;

  n.setConst(true);

  l.setConst(true);

  addInput("t", &t);
  addInput("t2", &t2);
  addInput("c", &c);
  addInput("n", &n);
  addInput("l", &l);
  addOutput("areaOut", &areaOut);
}

mvar evaluate(){
  try{
    double error = 0;

    for(size_t i = 0; i < 10; ++i){
      double length = uniform(1, 100);
```

```
      l = length;

      for(size_t ni = 3; ni < 10; ++ni){
        n = ni;

        double theta = 2*M_PI/ni;
        t = theta;
        t2 = theta/2;

        double rv = length/(2*sin(theta/2));

        double da = 0.5*ni*rv*rv*sin(theta);

        if(!run()){
          return undef;
        }

        double ei = areaOut.val() - da;
        ei *= ei;
        error += ei;
      }
    }

    if(error < 0.0001){
      return mvar::inf();
    }
```

```
      return 1/error;
    }
    catch(MError& e){
      return undef;
    }
  }

private:
  Length l;
  Angle t;
  Angle t2;
  Count c;
  Count n;
  Area areaOut;
};

int main(int argc, char** argv){
  Program program(argc, argv);

  program.init();
  program.start();

  return 0;
}
```

Appendix A5: Mancala Program Setup

```
#include <iostream>

#include <cmath>

#include <Meta/MLib.h>
#include <Meta/MRandom.h>

#include "MCProgram.h"
#include "Mancala.h"
#include "Count.h"
#include "Index.h"
#include "Truth.h"

using namespace std;
using namespace Meta;
using namespace Func;

static const size_t MATCHES = 50;

class Program : public MCProgram{
public:
  Program(int argc, char** argv)
    : MCProgram(argc, argv){

  }

  void setup(){
```

```
    mancala.setConst(true);

    slot.setConst(true);
    slot.setWeight(100);

    index0.setConst(true);
    index0.setStatic(true);
    index0 = 0;

    index1.setConst(true);
    index1.setStatic(true);
    index1 = 1;

    index2.setConst(true);
    index2.setStatic(true);
    index2 = 2;

    index3.setConst(true);
    index3.setStatic(true);
    index3 = 3;

    index4.setConst(true);
    index4.setStatic(true);
    index4 = 4;

    index5.setConst(true);
    index5.setStatic(true);
    index5 = 5;
```

```
    index6.setConst(true);
    index6.setStatic(true);
    index6 = 6;

    index7.setConst(true);
    index7.setStatic(true);
    index7 = 7;

    index8.setConst(true);
    index8.setStatic(true);
    index8 = 8;

    index9.setConst(true);
    index9.setStatic(true);
    index9 = 9;

    index10.setConst(true);
    index10.setStatic(true);
    index10 = 10;

    index11.setConst(true);
    index11.setStatic(true);
    index11 = 11;

    index12.setConst(true);
    index12.setStatic(true);
    index12 = 12;
```

```
    count0.setConst(true);
    count0.setStatic(true);
    count0 = 0;

    count1.setConst(true);
    count1.setStatic(true);
    count1 = 1;

    addInput("slot", &slot);
    addInput("index0", &index0);
    addInput("index1", &index1);
    addInput("index2", &index2);
    addInput("index3", &index3);
    addInput("index4", &index4);
    addInput("index5", &index5);
    addInput("index6", &index6);
    addInput("index7", &index7);
    addInput("index8", &index8);
    addInput("index9", &index9);
    addInput("index10", &index10);
    addInput("index11", &index11);
    addInput("index12", &index12);
    addInput("count0", &count0);
    addInput("count1", &count1);
    addInput("mancala", &mancala);
    addOutput("truthOut", &truthOut);
}
```

```
mvar evaluate(){
  MRandom random;
  random.timeSeed();

  try{
    size_t totalFitness = 0;

    for(size_t i = 0; i < MATCHES; ++i){
      mancala.reset();

      bool done = false;

      for(;;){
        double mt = mvar::negInf();
        size_t mj;

        for(size_t j = 1; j <= 6; ++j){
          if(mancala.numPieces(true, j) > 0){
            slot = j;
            if(!run()){
              return undef;
            }
            double t = truthOut.val();
            if(t > mt){
              mt = t;
              mj = j;
            }
```

```
        }
      }

      int r = mancala.move(true, mj);

      if(r == 2){
        done = true;
        break;
      }
      else if(r == 0){
        for(;;){
          size_t j;

          for(;;){
            j = random.equilikely(1, 6);

            if(mancala.numPieces(false, j) > 0){
              break;
            }
          }

          int r = mancala.move(false, j);

          if(r == 2){
            done = true;
            break;
          }
          else if(r == 0){
```

```
                break;
              }
            }
          }

          if(done){
            break;
          }
        }

        totalFitness += mancala.score(true);
      }

      return double(totalFitness)/MATCHES;
    }
    catch(MError& e){
      return undef;
    }
  }

private:
  Index slot;
  Index index0;
  Index index1;
  Index index2;
  Index index3;
  Index index4;
  Index index5;
```

```
  Index index6;
  Index index7;
  Index index8;
  Index index9;
  Index index10;
  Index index11;
  Index index12;
  Count count0;
  Count count1;
  Truth truthOut;
  Mancala mancala;
};

int main(int argc, char** argv){
  Program program(argc, argv);

  program.init();
  program.start();

  return 0;
}
```

Appendix A5: Ontology Headers

Angle.h:

```
#ifndef ANGLE_H
#define ANGLE_H

#include "Real.h"

namespace Meta{

class Length;

/*[
  description: "Angle",
  enabled: true
]*/
class Angle : public Real{
public:
  Angle();

  Angle(ConceptType p, const mvar& metadata);

  Angle(const Angle& c);

  virtual ~Angle();

  virtual Angle* copy() const{
    return new Angle(*this);
  }

  virtual Angle* create() const{
```

```
  return new Angle;
}

Angle& operator=(const mvar& v){
  set(v);
  return *this;
}

virtual bool handle(mnode n, uint32_t flags);

virtual mstr name() const{
  return "Angle";
}

virtual void validate(const mvar& v);

static mvar metadata();

/*[
  self: [vec:undef],
  ret: [ratio:true],
  post: {
    if(self.hasKey("vec")){
      ret.vec = true;
    }
  }
]*/
Length* sin() const;
```

```
  /*[
    self: [vec:undef],
    ret: [ratio:true],
    post: {
      if(self.hasKey("vec")){
        ret.vec = true;
      }
    }
  ]*/
  Length* cos() const;

private:
  class Angle_* x_;
};

} // end namespace Meta

#endif // ANGLE_H
```

Area.h:

```
#ifndef AREA_H
#define AREA_H

#include "Real.h"

namespace Meta{
```

```
class Volume;
class Length;

/*[
  description: "Area",
  description: "Area2"
]*/
class Area : public Real{
public:
  Area();

  Area(ConceptType p, const mvar& metadata);

  Area(const Area& c);

  virtual ~Area();

  virtual Area* copy() const{
    return new Area(*this);
  }

  virtual Area* create() const{
    return new Area;
  }

  Area& operator=(const mvar& v){
    set(v);
```

```
    return *this;
  }

  virtual bool handle(mnode n, uint32_t flags);

  virtual mstr name() const{
    return "Area";
  }

  virtual void validate(const mvar& v);

  /*[
    self: [vec:undef],
    l: [ratio:true],
    post: {
      if(l.hasKey("vec") || self.hasKey("vec")){
        ret.vec = true;
      }
    }
  ]*/
  Area* mulLengthRatio(const Length* l) const;

  static mvar metadata();

private:
  class Area_* x_;
};
```

```
} // end namespace Meta

#endif // AREA_H
```

Count.h:

```
#ifndef COUNT_H
#define COUNT_H

#include "Integer.h"

namespace Meta{

/*[
  description: "Count",
  enabled: true
]*/
class Count : public Integer{
public:
  Count();

  Count(ConceptType p, const mvar& metadata);

  Count(const Count& c);

  virtual ~Count();

  virtual Count* copy() const{
```

```
    return new Count(*this);
  }

  virtual Count* create() const{
    return new Count;
  }

  Count& operator=(const mvar& v){
    set(v);
    return *this;
  }

  virtual bool handle(mnode n, uint32_t flags);

  virtual mstr name() const{
    return "Count";
  }

  virtual void validate(const mvar& v);

  static mvar metadata();

private:
  class Count_* x_;
};

} // end namespace Meta
```

```
#endif // COUNT_H
```

Index.h:

```
#ifndef INDEX_H
#define INDEX_H

#include "Count.h"

namespace Meta{

/*[
  description: "Index",
  enabled: true
]*/
class Index : public Count{
public:
  Index();

  Index(ConceptType p, const mvar& metadata);

  Index(const Index& c);

  virtual ~Index();

  virtual Index* copy() const{
    return new Index(*this);
  }
```

```
  virtual Index* create() const{
    return new Index;
  }

  Index& operator=(const mvar& v){
    set(v);
    return *this;
  }

  virtual bool handle(mnode n, uint32_t flags);

  virtual mstr name() const{
    return "Index";
  }

  static mvar metadata();

private:
  class Index_* x_;
};

} // end namespace Meta

#endif // INDEX_H
```

Length.h:

```
#ifndef LENGTH_H
#define LENGTH_H

#include "Real.h"

#include <iostream>

namespace Meta{

class Area;

/*[
  description: "Length",
  enabled: true
]*/
class Length : public Real{
public:
  Length();

  Length(ConceptType p, const mvar& metadata);

  Length(const Length& c);

  virtual ~Length();

  virtual Length* copy() const{
    return new Length(*this);
```

```
}

virtual Length* create() const{
  return new Length;
}

Length& operator=(const mvar& v){
  set(v);
  return *this;
}

virtual bool handle(mnode n, uint32_t flags);

virtual mstr name() const{
  return "Length";
}

/*[
  self: [vec:undef],
  l: [vec:undef, takeThis:true],
  post: {
    if(l.hasKey("vec") || self.hasKey("vec")){
      ret.vec = true;
    }
  }
]*/
Area* mulToArea(const Length* l) const;
```

```
  virtual void validate(const mvar& v);

  static mvar metadata();

private:
  class Length_* x_;
};

} // end namespace Meta

#endif // LENGTH_H
```

Mancala.h:

```
#ifndef MANCALA_H
#define MANCALA_H

#include "Concept.h"

namespace Meta{

class Index;
class Count;
class Truth;

/*[
  description: "Mancala",
  enabled: true
```

```
]*/
class Mancala : public Concept{
public:
  Mancala();

  Mancala(ConceptType p, const mvar& metadata);

  Mancala(const Mancala& c);

  virtual ~Mancala();

  virtual Mancala* copy() const{
    return new Mancala(*this);
  }

  virtual Mancala* create() const{
    return new Mancala;
  }

  Mancala& operator=(const mvar& v){
    set(v);
    return *this;
  }

  virtual bool handle(mnode n, uint32_t flags);

  virtual mstr name() const{
    return "Mancala";
```

```
}

virtual void validate(const mvar& v);

static mvar metadata();

virtual mvar val() const final;

virtual void set(const mvar& v) final;

void reset();

int move(bool first, size_t slot);

size_t score(bool first) const;

size_t numPieces(bool first, size_t slot);

void writeBoard() const;

/*[
  enabled: true
]*/
Count* numPieces(const Index* i) const;

/*[
  enabled: true
]*/
```

```
Index* slotOffset(const Index* i, const Count* c) ↩
    const;

/*[
  enabled: true
]*/
Count* slotDiff(const Index* a, const Index* b) const;

/*[
  enabled: true
]*/
Truth* isMine(const Index* i) const;

/*[
  enabled: true
]*/
Truth* truthAnd(const Truth* a, const Truth* b) const;

/*[
  enabled: true
]*/
Truth* truthOr(const Truth* a, const Truth* b) const;

/*[
  enabled: true
]*/
Truth* truthNot(const Truth* t) const;
```

```
/*[
  enabled: true
]*/
Truth* countIndexEqual(const Count* c, const Index* i)↩
    const;

/*[
  enabled: true
]*/
Truth* countIndexGreater(const Count* c, const Index* ↩
   i) const;

/*[
  enabled: true
]*/
Truth* countIndexLesser(const Count* c, const Index* i↩
   ) const;

/*[
  enabled: true
]*/
Truth* countEqual(const Count* a, const Count* b) ↩
   const;

/*[
  enabled: true
]*/
```

```
  Truth* countGreater(const Count* a, const Count* b) ↩
     const;

  /*[
    enabled: true
  ]*/
  Truth* countLesser(const Count* a, const Count* b) ↩
     const;

  /*[
    enabled: true
  ]*/
  Count* countAdd(const Count* a, const Count* b) const;

  /*[
    enabled: true
  ]*/
  Count* countSub(const Count* a, const Count* b) const;

private:
  class Mancala_* x_;
};

} // end namespace Meta

#endif // MANCALA_H
```

Real.h:

```
#ifndef REAL_H
#define REAL_H

#include "Concept.h"

namespace Meta{

class Count;

/*[
  description: "Real"
]*/
class Real : public Concept{
public:
  Real();

  Real(const Real& c);

  Real(ConceptType p, const mvar& metadata);

  virtual ~Real();

  virtual Real* copy() const{
    return new Real(*this);
  }

  virtual Real* create() const{
```

```
  return new Real;
}

Real& operator=(const mvar& v){
  set(v);
  return *this;
}

virtual bool handle(mnode n, uint32_t flags);

virtual mstr name() const{
  return "Real";
}

virtual mvar val() const final;

virtual void set(const mvar& v) final;

virtual void validate(const mvar& v);

static mvar metadata();

virtual double match(const mvar& v, bool full);

void setDelta(bool flag);

bool getDelta() const;
```

```
  void undefDelta();

  void setRatio(bool flag);

  bool getRatio() const;

  void undefRatio();

  void setMultiplier(bool flag);

  bool getMultiplier() const;

  void undefMultiplier();

  void setCoord(bool flag);

  bool getCoord() const;

  void undefCoord();

  /*[
    self: [vec:true]
  ]*/
  Concept* sum() const;

  /*[
    self: [vec:true]
  ]*/
```

```
Concept* mean() const;

/*[
  self: [vec:true]
]*/
Concept* max() const;

/*[
  self: [vec:true]
]*/
Concept* min() const;

/*[
  self: [set:true, vec:true],
  c: [set:true, vec:true],
  ret: [set:true, vec:true]
]*/
Concept* setUnion(const Concept* c) const;

/*[
  self: [set:true, vec:true],
  c: [set:true, vec:true],
  ret: [set:true, vec:true]
]*/
Concept* setIntersection(const Concept* c) const;

/*[
  self: [set:true, vec:true],
```

```
    c: [set:true, vec:true],
    ret: [set:true, vec:true]
  ]*/
  Concept* setComplement(const Concept* c) const;

  /*[
    enabled: true,
    self: [set:true, vec:true]
  ]*/
  void foo();

  /*[
    enabled: true,
    self: [set:false, vec:false]
  ]*/
  void bar();

  /*[
    self: [vec:undef],
    c: [vec:undef, takeThis:true],
    post: {
      if(c.hasKey("vec") || self.hasKey("vec")){
        ret.vec = true;
      }
    }
  ]*/
  Concept* add(const Concept* c) const;
```

```
/*[
  self: [vec:true]
]*/
Count* length() const;

/*[
  self: [vec:undef, ratio:undef],
  count: [vec:undef],
  post: {
    if(count.hasKey("vec") || self.hasKey("vec")){
      ret.vec = true;
    }

    if(self.hasKey("ratio")){
      ret.ratio = true;
    }
  }
]*/
Concept* mulCount(const Count* count) const;

/*[
  self: [vec:undef, ratio:undef],
  count: [vec:undef],
  post: {
    if(count.hasKey("vec") || self.hasKey("vec")){
      ret.vec = true;
    }
```

```
    if(self.hasKey("ratio")){
      ret.ratio = true;
    }
  }
]*/
Concept* divCount(const Count* count) const;

/*[
  self: [vec:undef, ratio:undef],
  r: [ratio:true],
  post: {
    if(r.hasKey("vec") || self.hasKey("vec")){
      ret.vec = true;
    }
  }
]*/
Concept* mulRatio(const Concept* r) const;

/*[
  self: [vec:undef],
  r: [ratio:true],
  post: {
    if(r.hasKey("vec") || self.hasKey("vec")){
      ret.vec = true;
    }
  }
]*/
Concept* divRatio(const Concept* r) const;
```

```
  /*[
    self: [vec:undef],
    c: [vec:undef],
    ret: [ratio:true],
    post: {
      if(c.hasKey("vec") || self.hasKey("vec")){
        ret.vec = true;
      }
    }
  ]*/
  Concept* divToRatio(const Concept* c) const;

private:
  class Real_* x_;
};

} // end namespace Meta

#endif // REAL_H
```

Truth.h:

```
#ifndef TRUTH_H
#define TRUTH_H

#include "Concept.h"
```

```
namespace Meta{

/*[
  description: "Truth",
  enabled: true
]*/
class Truth : public Concept{
public:
  Truth();

  Truth(ConceptType p, const mvar& metadata);

  Truth(const Truth& c);

  virtual ~Truth();

  virtual Truth* copy() const{
    return new Truth(*this);
  }

  virtual Truth* create() const{
    return new Truth;
  }

  Truth& operator=(const mvar& v){
    set(v);
    return *this;
  }
```

```
virtual bool handle(mnode n, uint32_t flags);

virtual mstr name() const{
  return "Truth";
}

virtual void validate(const mvar& v);

static mvar metadata();

virtual mvar val() const final;

virtual void set(const mvar& v) final;

/*[
  enabled: true
]*/
Truth* truthAnd(const Truth* t) const;

/*[
  enabled: true
]*/
Truth* truthOr(const Truth* t) const;

/*[
  enabled: true
]*/
```

```
  Truth* truthNot() const;

private:
  class Truth_* x_;

  mvar value_;
};

} // end namespace Meta

#endif // TRUTH_H
```

Volume.h:

```
#ifndef VOLUME_H
#define VOLUME_H

#include "Real.h"

namespace Meta{

/*[
  description: "Volume"
]*/
class Volume : public Real{
public:
  Volume();
```

```
Volume(ConceptType p, const mvar& metadata);

Volume(const Volume& c);

virtual ~Volume();

virtual Volume* copy() const{
  return new Volume(*this);
}

virtual Volume* create() const{
  return new Volume;
}

Volume& operator=(const mvar& v){
  set(v);
  return *this;
}

virtual bool handle(mnode n, uint32_t flags);

virtual mstr name() const{
  return "Volume";
}

virtual void validate(const mvar& v);

static mvar metadata();
```

```
private:
  class Volume_* x_;
};

} // end namespace Meta

#endif // VOLUME_H
```

Appendix A6: Additional Area Solutions, Metrics, and Plots

Solution 2C:

```
----- round:            2
----- iterations:       37965
----- run time:         10912.7
----- round time:       2360.16
----- population size:  3000
----- best fitness:     4.79257e-08
----- worst fitness:    2.89595e-10
----- avg. fitness:     1.29006e-09
----- avg. temps:       10.3697
----- avg. norm. size:  12.3697
----- avg. orig. size:  49.3783
----- temp util:        1.19287
----- avg. reduction:   4.22728
```

```
----- total solutions:  59310
----- total misses:     15147
----- hit rate:         0.744613
----- total accepted:   6684
----- total denied:     24815
----- acceptance rate:  0.269353
----- total errors:     6197
----- error rate:       0.104485
----- total duplicates: 11705
----- duplication rate: 0.30831

----- trial:      1/1
----- rounds: 3
----- iterations: 37965
----- final solution
Length* length_0 = t2.sin();
Length* length_1 = l.divRatio(length_0);
Length* length_2 = t2.sin();
Length* length_3 = length_1.divCount(c);
Length* length_4 = length_3.divCount(c);
Length* length_5 = length_2.divCount(c);
Length* length_6 = length_4.divRatio(length_5);
Area* area_7 = length_3.mulToArea(length_6);
Area* area_8 = area_7.mulLengthRatio(length_5);
Length* length_9 = t.sin();
Length* length_10 = length_9.mulCount(n);
Area* area_11 = area_8.mulLengthRatio(length_10);
areaOut.setObj(area_11);
```

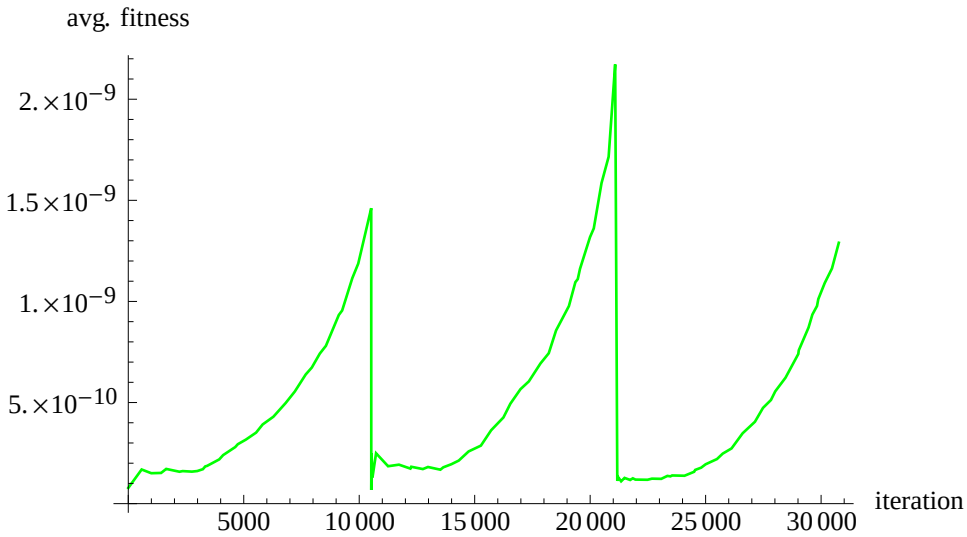


Figure 7.1: Run 2C avg. fitness

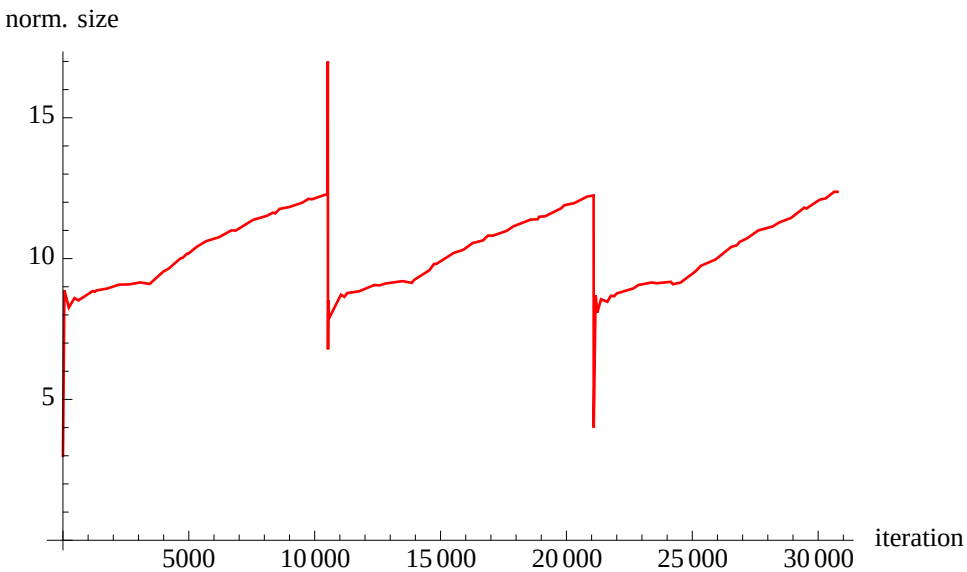


Figure 7.2: Run 2C norm. size

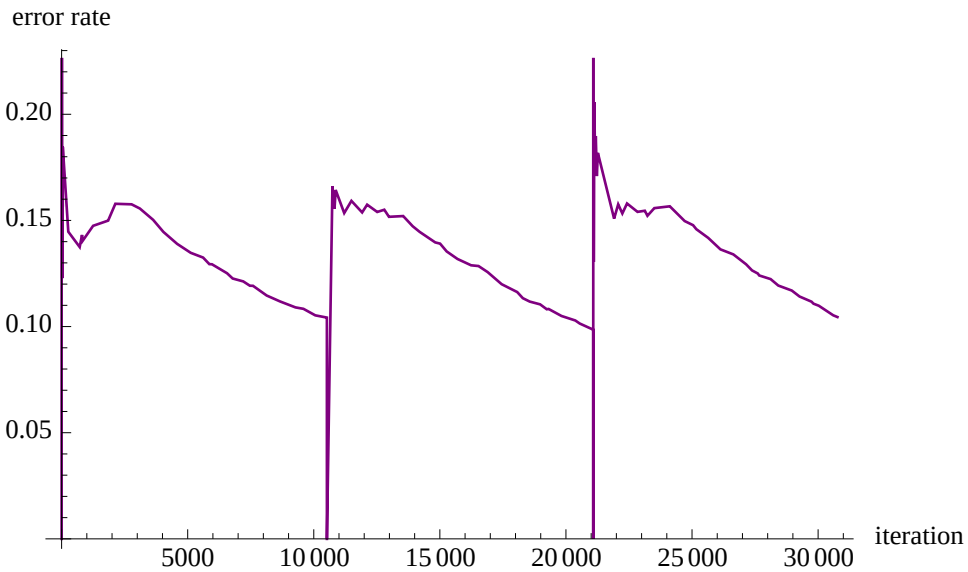


Figure 7.3: Run 2C error rate

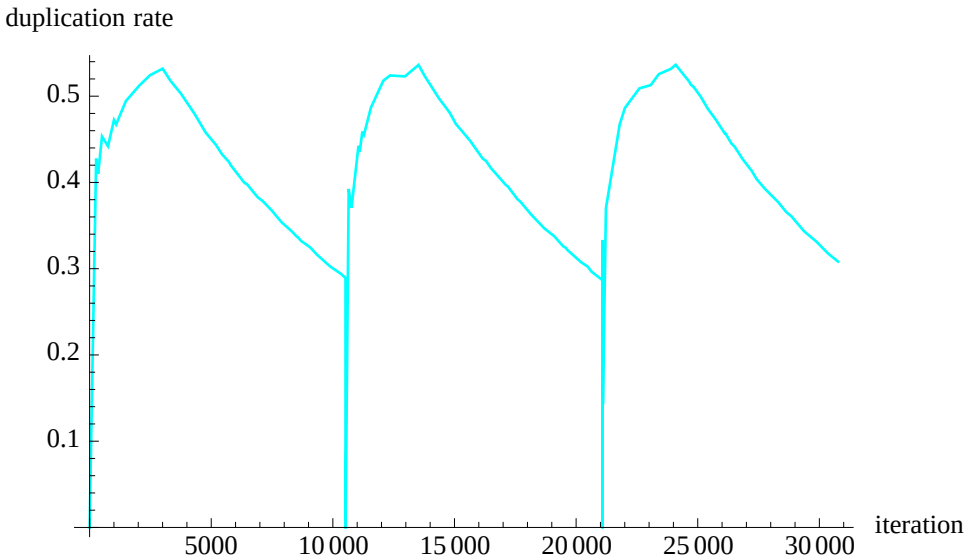


Figure 7.4: Run 2C duplication rate

Solution 2D:

```
----- round:            0
----- iterations:       42139
----- run time:         4772.34
----- round time:       4772.34
----- population size:  3000
----- best fitness:     6.61481e-08
----- worst fitness:    3.33262e-10
----- avg. fitness:     1.48222e-09
----- avg. temps:       10.3173
----- avg. norm. size:  12.3173
----- avg. orig. size:  50.6763
----- temp util:        1.19385
----- avg. reduction:   4.32621
----- total solutions:  66820
----- total misses:     17570
----- hit rate:         0.737055
----- total accepted:   7090
----- total denied:     28694
----- acceptance rate:  0.24709
----- total errors:     7110
----- error rate:       0.106405
----- total duplicates: 12918
----- duplication rate: 0.306557

----- trial:      1/1
----- rounds: 1
```

```
----- iterations: 42139
----- final solution
Angle* angle_0 = t2.divCount(c);
Length* length_1 = l.divCount(c);
Angle* angle_2 = angle_0.mulCount(c);
Length* length_3 = angle_2.sin();
Length* length_4 = length_1.divRatio(length_3);
Length* length_5 = angle_0.sin();
Length* length_6 = length_4.divRatio(length_5);
Length* length_7 = length_4.divCount(c);
Area* area_8 = length_7.mulToArea(length_6);
Area* area_9 = area_8.mulCount(n);
Length* length_10 = angle_0.sin();
Area* area_11 = area_9.mulLengthRatio(length_10);
Length* length_12 = t.sin();
Area* area_13 = area_11.mulLengthRatio(length_12);
areaOut.setObj(area_13);
```

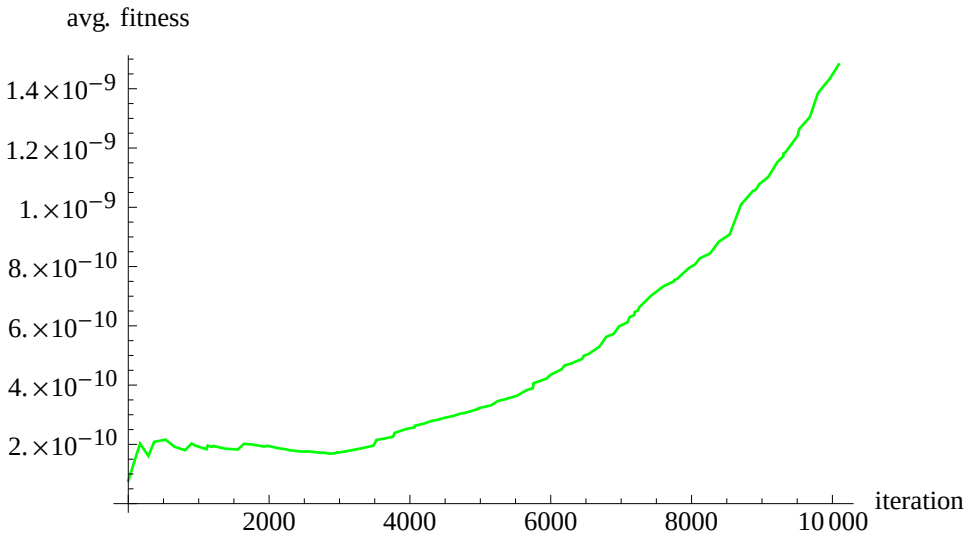


Figure 7.5: Run 2D avg. fitness

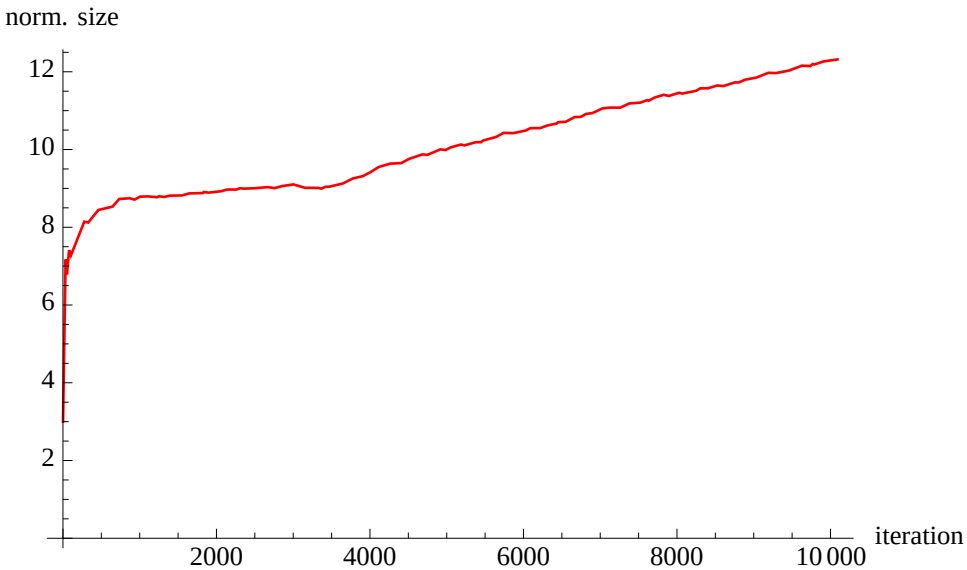


Figure 7.6: Run 2D norm. size

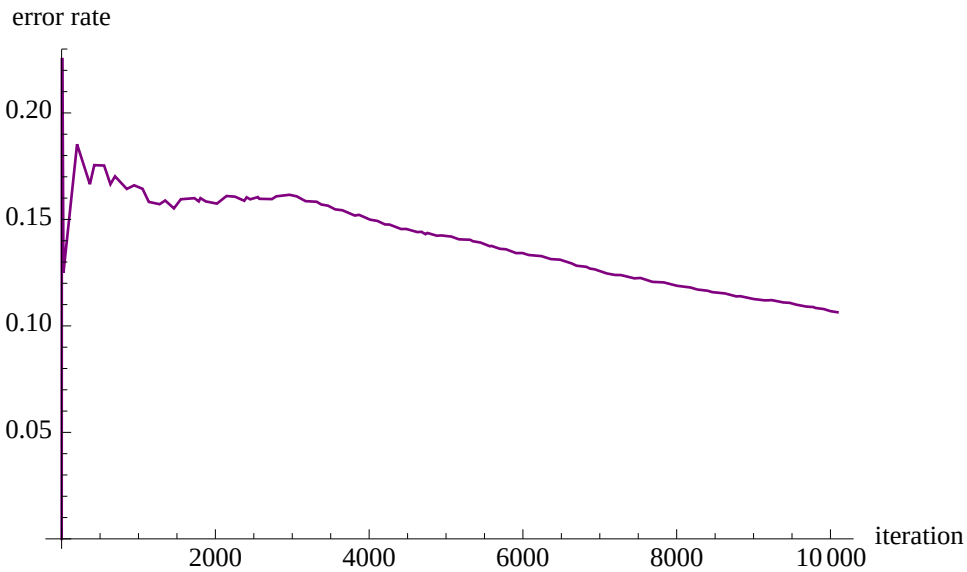


Figure 7.7: Run 2D error rate

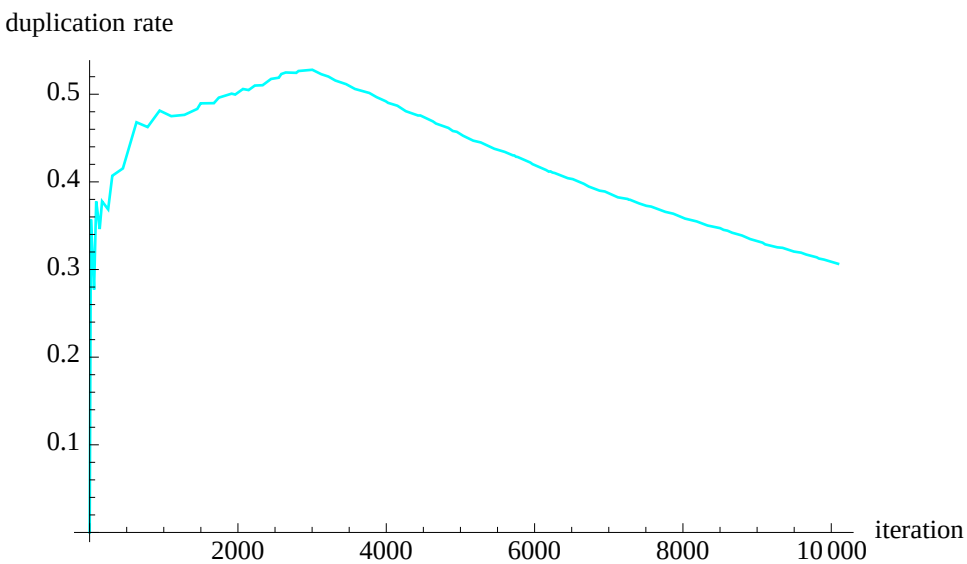


Figure 7.8: Run 2D duplication rate

Solution 2E:

```
----- round:            0
----- iterations:       26184
----- run time:         3323.39
----- round time:       3323.39
----- population size:  3000
----- best fitness:     3.14528e-08
----- worst fitness:    1.7475e-10
----- avg. fitness:     7.68365e-10
----- avg. temps:       9.55333
----- avg. norm. size:  11.5533
----- avg. orig. size:  48.0613
----- temp util:        1.20935
----- avg. reduction:   4.38942
----- total solutions:  42388
----- total misses:     11358
----- hit rate:         0.732047
----- total accepted:   5283
----- total denied:     14468
----- acceptance rate:  0.365151
----- total errors:     4845
----- error rate:       0.114301
----- total duplicates: 8919
----- duplication rate: 0.340628

----- trial:      1/1
----- rounds: 1
```

```
----- iterations: 26184
----- final solution
Length* length_0 = t.sin();
Length* length_1 = l.divCount(c);
Length* length_2 = length_1.divRatio(length_0);
Length* length_3 = t2.sin();
Length* length_4 = length_2.divRatio(length_3);
Area* area_5 = length_4.mulToArea(length_4);
Length* length_6 = t.sin();
Length* length_7 = length_6.divCount(c);
Length* length_8 = length_7.mulCount(c);
Area* area_9 = area_5.mulLengthRatio(length_8);
Length* length_10 = length_7.mulCount(c);
Length* length_11 = length_8.mulCount(n);
Area* area_12 = area_9.mulLengthRatio(length_11);
Area* area_13 = area_12.mulLengthRatio(length_10);
Area* area_14 = area_13.divCount(c);
Area* area_15 = area_14.mulCount(n);
Area* area_16 = area_15.divCount(n);
areaOut.setObj(area_16);
```

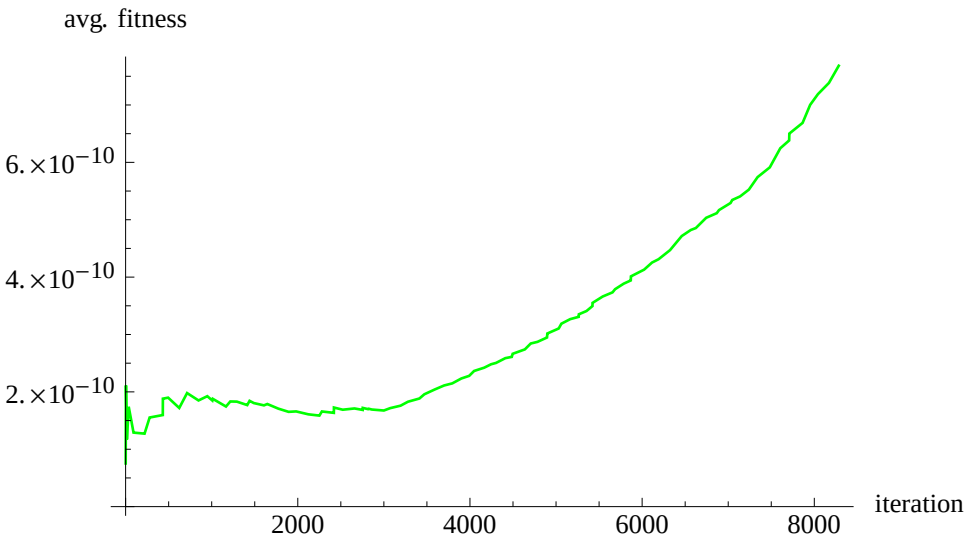


Figure 7.9: Run 2E avg. fitness

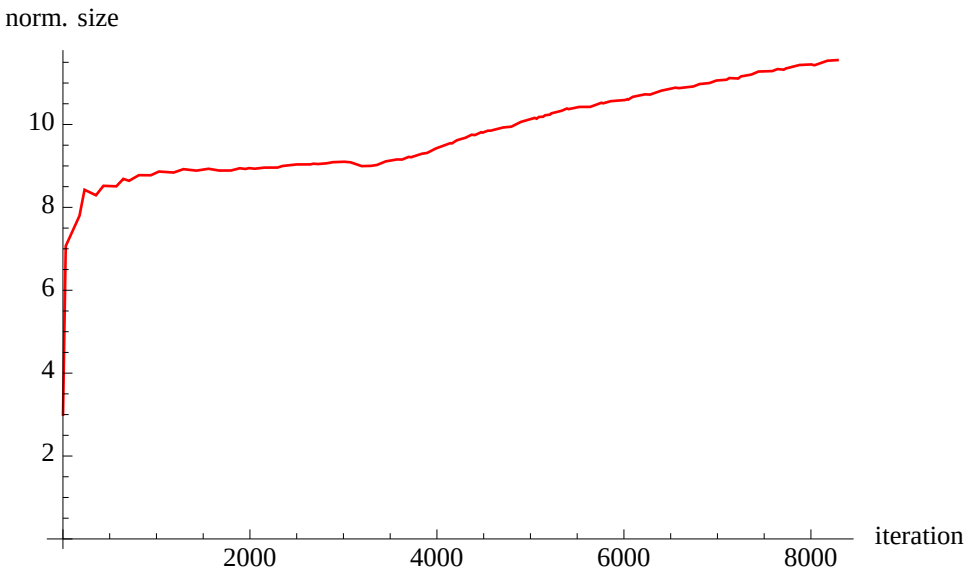


Figure 7.10: Run 2E norm. size

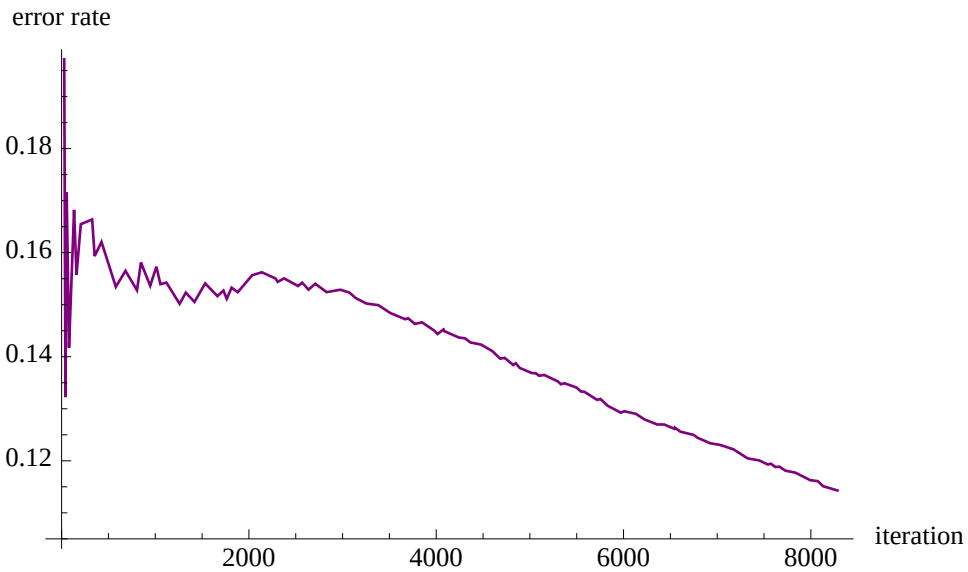


Figure 7.11: Run 2E error rate

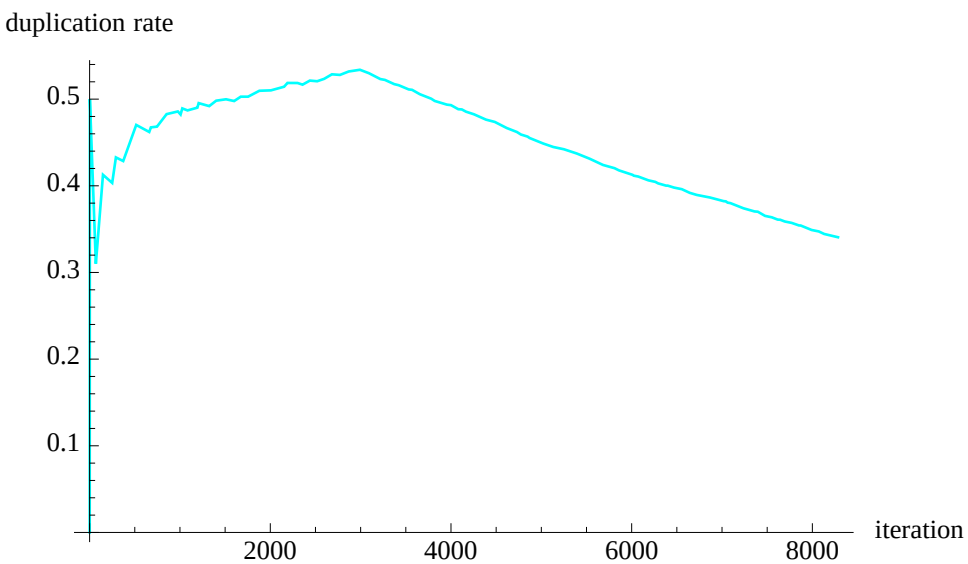


Figure 7.12: Run 2E duplication rate

Solution 2F:

```
----- round:            0
----- iterations:       34966
----- run time:         4326.44
----- round time:       4326.44
----- population size:  3000
----- best fitness:     9.59377e-08
----- worst fitness:    2.4818e-10
----- avg. fitness:     1.15196e-09
----- avg. temps:       10.0247
----- avg. norm. size:  12.0247
----- avg. orig. size:  49.2873
----- temp util:        1.19951
----- avg. reduction:   4.31465
----- total solutions:  55565
----- total misses:     14397
----- hit rate:         0.740898
----- total accepted:   6382
----- total denied:     22246
----- acceptance rate:  0.286883
----- total errors:     6201
----- error rate:       0.111599
----- total duplicates: 10944
----- duplication rate: 0.31299

----- trial:      1/1
----- rounds: 1
```

```
----- iterations: 34966
----- final solution
Length* length_0 = t2.sin();
Count* count_1 = n.mulCount(c);
Length* length_2 = l.mulCount(n);
Length* length_3 = length_2.divRatio(length_0);
Area* area_4 = length_3.mulToArea(length_3);
Length* length_5 = t.sin();
Count* count_6 = count_1.mulCount(c);
Area* area_7 = area_4.mulLengthRatio(length_5);
Count* count_8 = count_6.mulCount(c);
Area* area_9 = area_7.divCount(count_8);
areaOut.setObj(area_9);
```

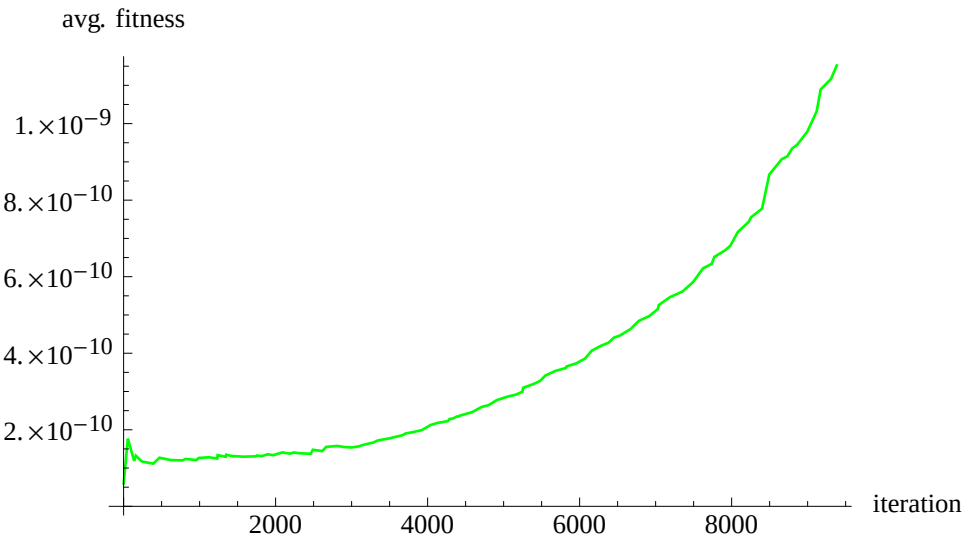


Figure 7.13: Run 2F avg. fitness

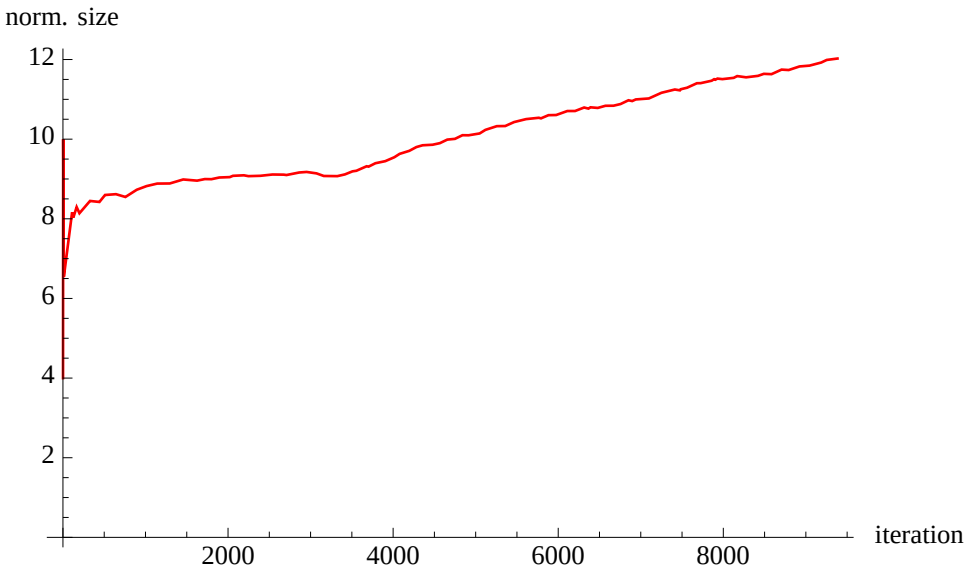


Figure 7.14: Run 2F norm. size

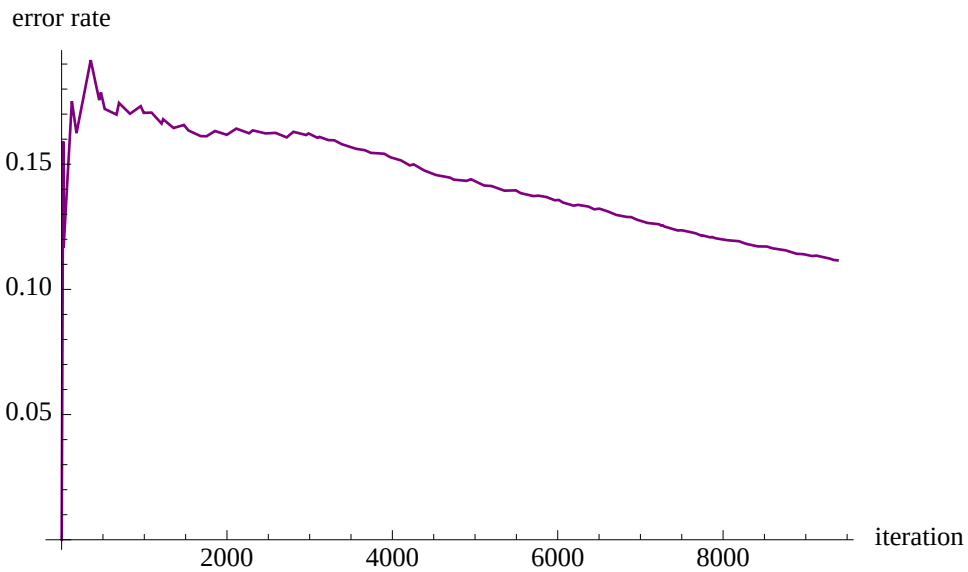


Figure 7.15: Run 2F error rate

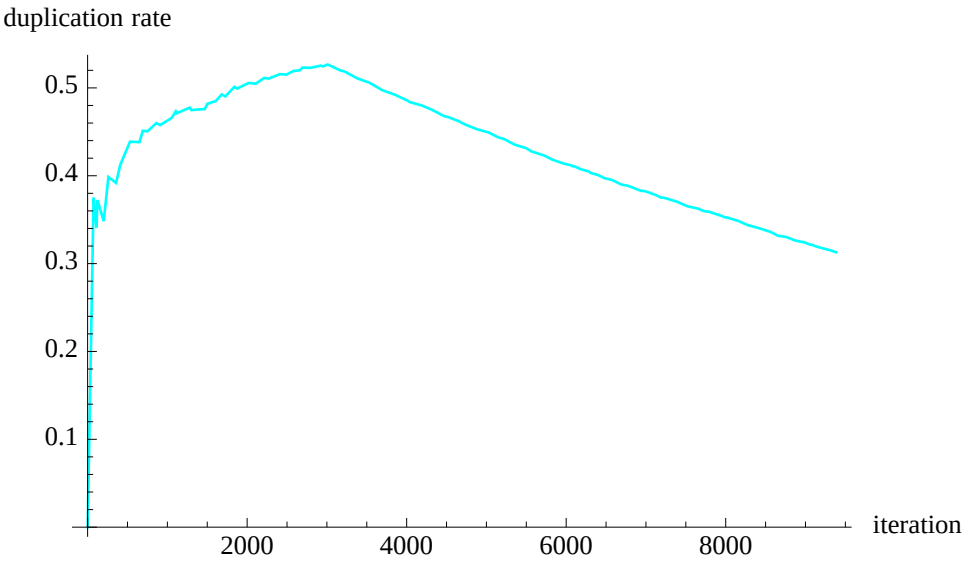


Figure 7.16: Run 2F duplication rate

Solution 2G:

```
----- round:            1
----- iterations:       44172
----- run time:         8219.28
----- round time:       2844.13
----- population size:  3000
----- best fitness:     4.2453e-08
----- worst fitness:    3.48053e-10
----- avg. fitness:     1.41678e-09
----- avg. temps:       10.3917
----- avg. norm. size:  12.3917
----- avg. orig. size:  50.4203
----- temp util:        1.19246
----- avg. reduction:   4.29565
----- total solutions:  68004
----- total misses:     16830
----- hit rate:         0.752515
----- total accepted:   7205
----- total denied:     30514
----- acceptance rate:  0.236121
----- total errors:     7001
----- error rate:       0.10295
----- total duplicates: 13039
----- duplication rate: 0.295187

----- trial:      1/1
----- rounds: 2
```

```
----- iterations: 44172
----- final solution
Length* length_0 = t2.sin();
Length* length_1 = t2.sin();
Length* length_2 = l.divCount(c);
Length* length_3 = length_2.divRatio(length_1);
Length* length_4 = length_3.divRatio(length_0);
Length* length_5 = t2.sin();
Length* length_6 = length_4.divRatio(length_0);
Area* area_7 = length_4.mulToArea(length_6);
Area* area_8 = area_7.mulLengthRatio(length_5);
Length* length_9 = t2.sin();
Area* area_10 = area_8.mulLengthRatio(length_9);
Angle* angle_11 = t2.mulCount(c);
Length* length_12 = angle_11.sin();
Area* area_13 = area_10.mulLengthRatio(length_9);
Length* length_14 = length_12.divCount(n);
Area* area_15 = area_13.mulLengthRatio(length_14);
Area* area_16 = area_15.mulCount(n);
Area* area_17 = area_16.mulCount(n);
Area* area_18 = area_17.divCount(c);
areaOut.setObj(area_18);
```

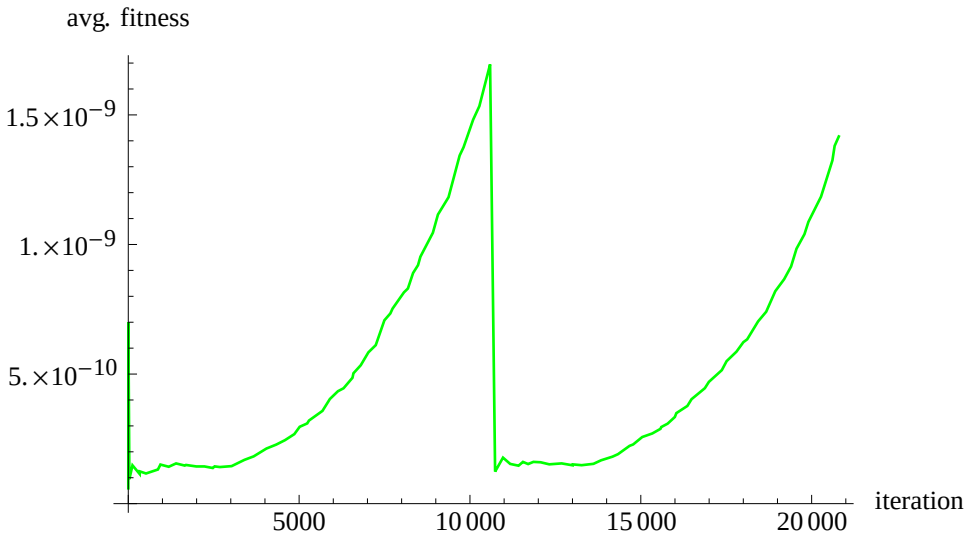


Figure 7.17: Run 2G avg. fitness

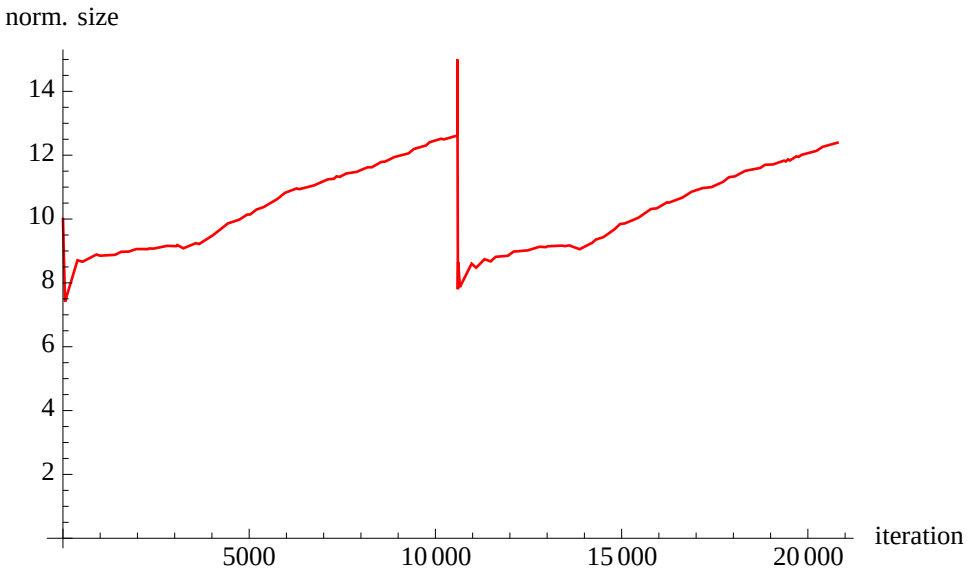


Figure 7.18: Run 2G norm. size

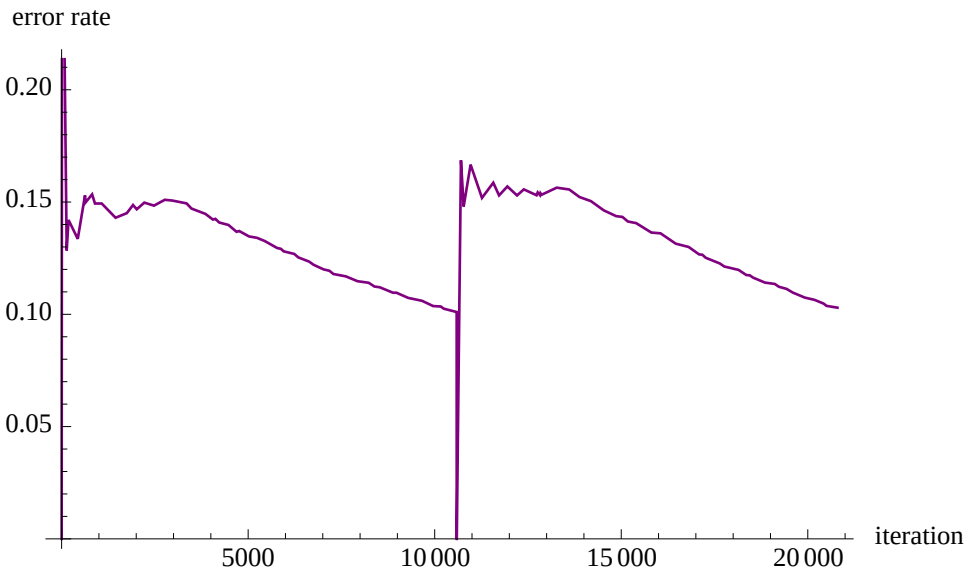


Figure 7.19: Run 2G error rate

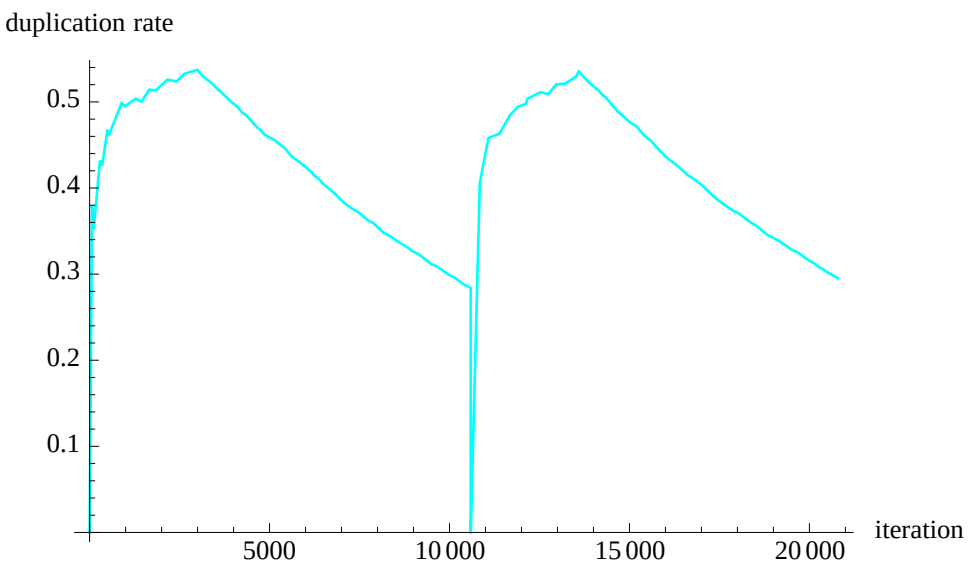


Figure 7.20: Run 2G duplication rate

Solution 2H:

```
----- round:            1
----- iterations:       44172
----- run time:         8219.28
----- round time:       2844.13
----- population size:  3000
----- best fitness:     4.2453e-08
----- worst fitness:    3.48053e-10
----- avg. fitness:     1.41678e-09
----- avg. temps:       10.3917
----- avg. norm. size:  12.3917
----- avg. orig. size:  50.4203
----- temp util:        1.19246
----- avg. reduction:   4.29565
----- total solutions:  68004
----- total misses:     16830
----- hit rate:         0.752515
----- total accepted:   7205
----- total denied:     30514
----- acceptance rate:  0.236121
----- total errors:     7001
----- error rate:       0.10295
----- total duplicates: 13039
----- duplication rate: 0.295187

----- trial:      1/1
----- rounds: 2
```

```
----- iterations: 44172
----- final solution
Length* length_0 = t2.sin();
Length* length_1 = t2.sin();
Length* length_2 = l.divCount(c);
Length* length_3 = length_2.divRatio(length_1);
Length* length_4 = length_3.divRatio(length_0);
Length* length_5 = t2.sin();
Length* length_6 = length_4.divRatio(length_0);
Area* area_7 = length_4.mulToArea(length_6);
Area* area_8 = area_7.mulLengthRatio(length_5);
Length* length_9 = t2.sin();
Area* area_10 = area_8.mulLengthRatio(length_9);
Angle* angle_11 = t2.mulCount(c);
Length* length_12 = angle_11.sin();
Area* area_13 = area_10.mulLengthRatio(length_9);
Length* length_14 = length_12.divCount(n);
Area* area_15 = area_13.mulLengthRatio(length_14);
Area* area_16 = area_15.mulCount(n);
Area* area_17 = area_16.mulCount(n);
Area* area_18 = area_17.divCount(c);
areaOut.setObj(area_18);
```

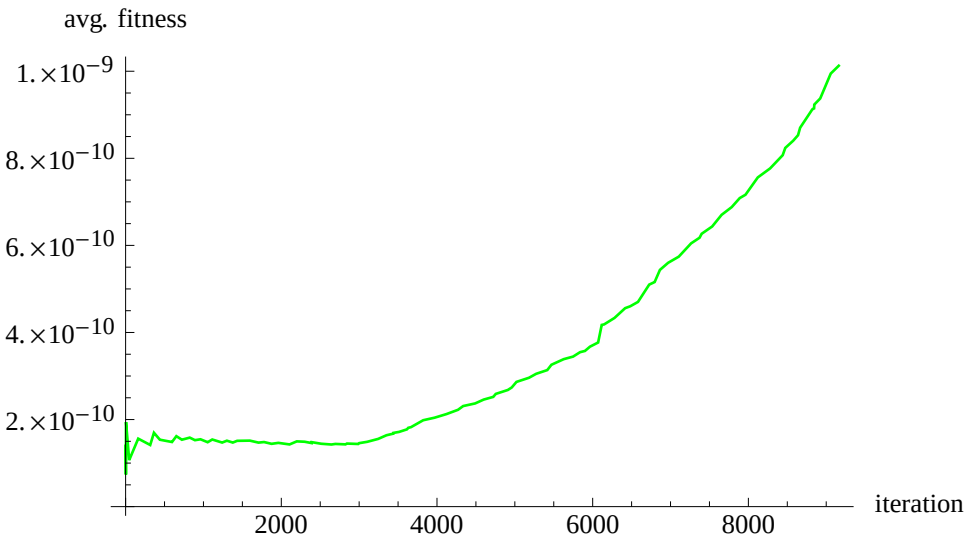


Figure 7.21: Run 2H avg. fitness

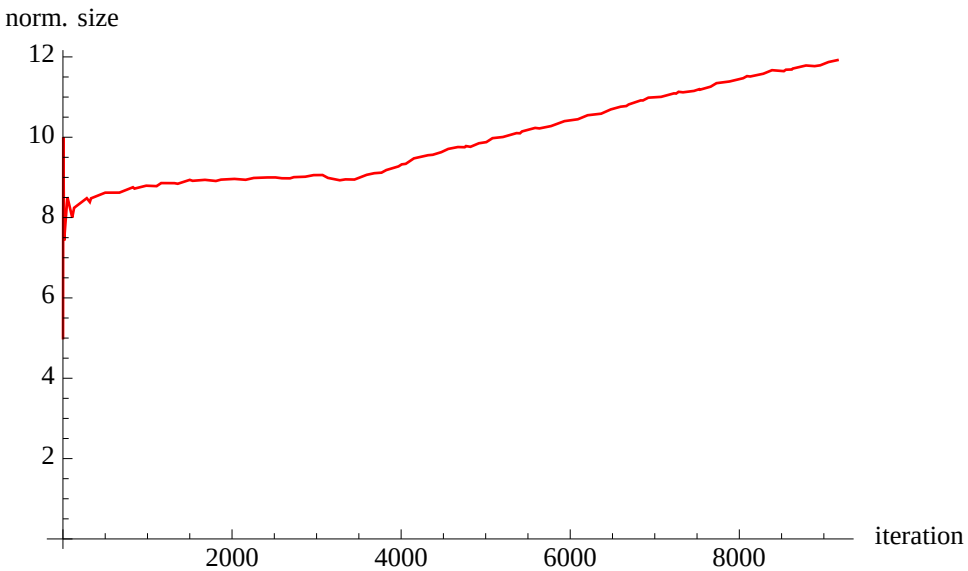


Figure 7.22: Run 2H norm. size

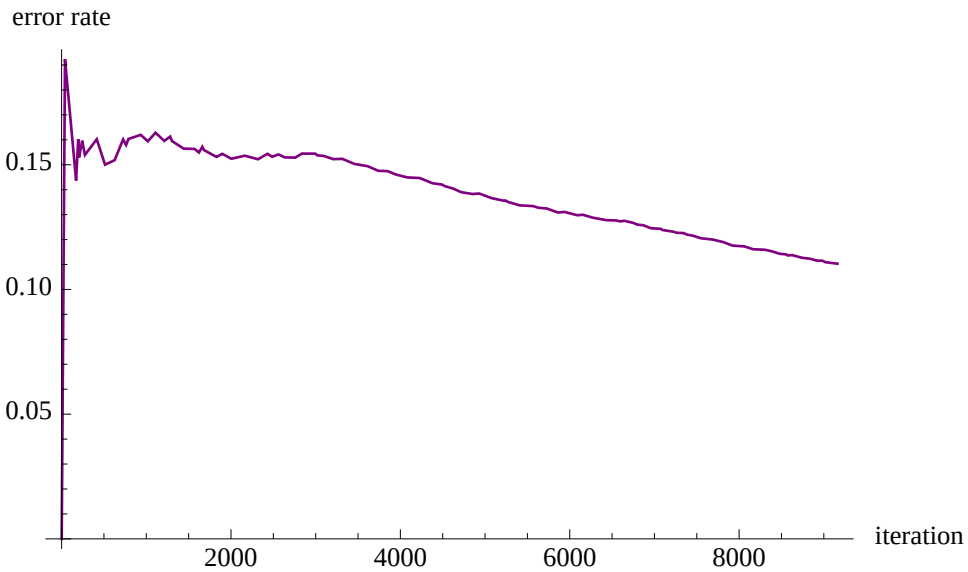


Figure 7.23: Run 2H error rate

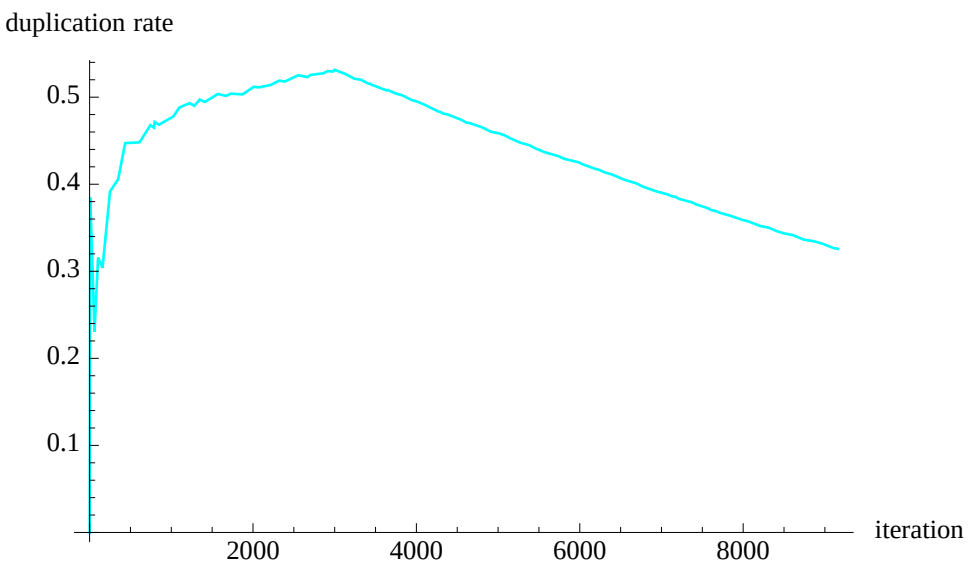


Figure 7.24: Run 2H duplication rate

Appendix A7: Mancala CGG

```
<<start>> => Mancala::countEqual
<<start>> => Mancala::countAdd
Mancala::countAdd => Mancala::countAdd
<<start>> => Mancala::countSub
Mancala::countAdd => Mancala::countSub
Mancala::countSub => Mancala::countSub
<<start>> => Mancala::numPieces
<<start>> => Mancala::slotOffset
Mancala::countAdd => Mancala::slotOffset
Mancala::countSub => Mancala::slotOffset
Mancala::numPieces => Mancala::slotOffset
<<start>> => Mancala::slotDiff
Mancala::slotOffset => Mancala::slotDiff
Mancala::slotDiff => Mancala::slotOffset
Mancala::slotOffset => Mancala::slotOffset
Mancala::slotOffset => Mancala::numPieces
Mancala::numPieces => Mancala::countSub
Mancala::slotDiff => Mancala::countSub
Mancala::countSub => Mancala::countAdd
Mancala::numPieces => Mancala::countAdd
Mancala::slotDiff => Mancala::countAdd
Mancala::countAdd => Mancala::countEqual
Mancala::countSub => Mancala::countEqual
Mancala::numPieces => Mancala::countEqual
Mancala::slotDiff => Mancala::countEqual
Mancala::countEqual => <<end>>
```

```
<<start>> => Mancala::countGreater
Mancala::countAdd => Mancala::countGreater
Mancala::countSub => Mancala::countGreater
Mancala::numPieces => Mancala::countGreater
Mancala::slotDiff => Mancala::countGreater
Mancala::countGreater => <<end>>
<<start>> => Mancala::countIndexEqual
Mancala::countAdd => Mancala::countIndexEqual
Mancala::countSub => Mancala::countIndexEqual
Mancala::numPieces => Mancala::countIndexEqual
Mancala::slotDiff => Mancala::countIndexEqual
Mancala::slotOffset => Mancala::countIndexEqual
Mancala::countIndexEqual => <<end>>
<<start>> => Mancala::countIndexGreater
Mancala::countAdd => Mancala::countIndexGreater
Mancala::countSub => Mancala::countIndexGreater
Mancala::numPieces => Mancala::countIndexGreater
Mancala::slotDiff => Mancala::countIndexGreater
Mancala::slotOffset => Mancala::countIndexGreater
Mancala::countIndexGreater => <<end>>
<<start>> => Mancala::countIndexLesser
Mancala::countAdd => Mancala::countIndexLesser
Mancala::countSub => Mancala::countIndexLesser
Mancala::numPieces => Mancala::countIndexLesser
Mancala::slotDiff => Mancala::countIndexLesser
Mancala::slotOffset => Mancala::countIndexLesser
Mancala::countIndexLesser => <<end>>
<<start>> => Mancala::countLesser
```

```
Mancala::countAdd => Mancala::countLesser
Mancala::countSub => Mancala::countLesser
Mancala::numPieces => Mancala::countLesser
Mancala::slotDiff => Mancala::countLesser
Mancala::countLesser => <<end>>
<<start>> => Mancala::isMine
Mancala::slotOffset => Mancala::isMine
Mancala::isMine => <<end>>
<<start>> => Mancala::truthAnd
Mancala::countEqual => Mancala::truthAnd
Mancala::countGreater => Mancala::truthAnd
Mancala::countIndexEqual => Mancala::truthAnd
Mancala::countIndexGreater => Mancala::truthAnd
Mancala::countIndexLesser => Mancala::truthAnd
Mancala::countLesser => Mancala::truthAnd
Mancala::isMine => Mancala::truthAnd
Mancala::truthAnd => Mancala::truthAnd
<<start>> => Mancala::truthNot
Mancala::countEqual => Mancala::truthNot
Mancala::countGreater => Mancala::truthNot
Mancala::countIndexEqual => Mancala::truthNot
Mancala::countIndexGreater => Mancala::truthNot
Mancala::countIndexLesser => Mancala::truthNot
Mancala::countLesser => Mancala::truthNot
Mancala::isMine => Mancala::truthNot
Mancala::truthAnd => Mancala::truthNot
Mancala::truthNot => Mancala::truthNot
<<start>> => Mancala::truthOr
```

```
Mancala::countEqual => Mancala::truthOr
Mancala::countGreater => Mancala::truthOr
Mancala::countIndexEqual => Mancala::truthOr
Mancala::countIndexGreater => Mancala::truthOr
Mancala::countIndexLesser => Mancala::truthOr
Mancala::countLesser => Mancala::truthOr
Mancala::isMine => Mancala::truthOr
Mancala::truthAnd => Mancala::truthOr
Mancala::truthNot => Mancala::truthOr
Mancala::truthOr => Mancala::truthOr
Mancala::truthOr => Mancala::truthNot
Mancala::truthNot => Mancala::truthAnd
Mancala::truthOr => Mancala::truthAnd
Mancala::truthAnd => <<end>>
Mancala::truthNot => <<end>>
Mancala::truthOr => <<end>>

total methods: 39
total nodes: 17
edge upper bound: 255
edges used: 98
unused edges: 157
edge usage: 0.384314
```