GENESIS: A LANGUAGE FOR GENERATING SYNTHETIC PROGRAMS

by

Alton Chiu

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2015 by Alton Chiu
Abstract

Genesis: A Language for Generating Synthetic Programs

Alton Chiu
Master of Applied Science
Graduate Department of Electrical and Computer Computer Engineering
University of Toronto
2015

This thesis describes Genesis, a new language used for the expression and generation of synthetic programs. The language allows users to annotate a template program to identify code segments of the program they wish to vary. Users also describe how parameters in each code segment can vary across generated instance programs using statistical distributions. The Genesis preprocessor uses the annotations to generate a set of instance programs whose characteristics or features vary in a statistically controlled fashion. We describe Genesis, its functionalities, its language constructs, a prototype preprocessor for the language, and three case studies that demonstrate the ability of Genesis to express a range of programs in different domains. We evaluate the performance of the preprocessor during the generation of instance programs, and the statistical quality of the samples it generates. We believe that Genesis is a useful tool that eases the expression and creation of large and diverse program sets for a multitude of domains and applications, such as machine learning, image generation, and compiler testing.
Acknowledgements

I would like to thank my supervisor, Prof. Tarek S. Abdelrahman, for his continual support, guidance, patience and encouragement. I would like to also thank my colleagues in our research group, notably Joseph Garvey and David Han, who have provided continual feedback and support on the work and thesis during the last three years.

I would like to thank my friends and family who have supported me throughout this process, and for putting up with my long nights and winded explanations of this thesis topic.

I would like to thank the Lewitt Family for their funding of the Queen Elizabeth II/Lewfam Scholarship in Science and Technology and their generosity and support.

This work was funded by research grants from NSERC and Qualcomm. I am thankful for their support.
## Contents

1 Introduction .................................................. 1
   1.1 Thesis Contribution ........................................ 3
   1.2 Thesis Organization ........................................ 4

2 Background .................................................... 5
   2.1 Program Generators ......................................... 5
      2.1.1 CSmith ................................................. 6
      2.1.2 TestMake .............................................. 7
      2.1.3 CodeSmith Generator .................................. 7
   2.2 Application Domains ........................................ 9
      2.2.1 Machine Learning ...................................... 9
      2.2.2 Compiler Testing ..................................... 11

3 Genesis Overview ............................................... 12
   3.1 The Flow of Genesis ....................................... 16
   3.2 A Second Example ......................................... 19

4 Genesis Constructs ............................................. 22
   4.1 The `distribution` Construct ............................. 22
   4.2 The `value` Construct .................................... 23
   4.3 The `varlist` Construct .................................. 24
4.4 The **variable** Construct ............................................. 26
4.5 The **feature** Construct ............................................. 26
  4.5.1 The **feature** Definition ........................................ 26
  4.5.2 **feature** Usage .................................................. 27
  4.5.3 Multiple **feature** References ................................. 27
  4.5.4 **feature** Arguments ............................................ 28
  4.5.5 Stored **features** ................................................ 28
4.6 The **generate** Construct ........................................... 29
4.7 The **genmath** Construct ........................................... 30
4.8 The **add** and **remove** Constructs ............................... 30
4.9 The **genif** Construct .............................................. 31
4.10 The **genloop** Construct .......................................... 32
4.11 The **geninclude** Construct ..................................... 33
4.12 The **genassert** Construct ...................................... 33

5 **Genesis Examples** .................................................. 35
  5.1 Using Variables ..................................................... 35
  5.2 Using **enumerate** .................................................. 37
  5.3 Using **genif** ....................................................... 42
  5.4 Using **genloop** ..................................................... 44
  5.5 Using Processing Order .......................................... 46
  5.6 Using the Sections of Genesis .................................. 49
  5.7 Using **enumerate** with Stored Features ....................... 50
  5.8 Using Multiple **enumerate** Statements ....................... 52
  5.9 Using Library Files .............................................. 55

6 **Case Studies** ....................................................... 58
  6.1 Case Study 1: Image Filtering ................................... 58
## List of Tables

2.1 Example Student Information ........................................ 8

8.1 Test Results for One Run of the Image Filtering Genesis Program .... 83

8.2 Test Results for Value Distributions .................................. 84

8.3 Breakdown of Program Generation Time for Image Filtering .......... 86

A.1 Table of Command Line Arguments .................................... 97

A.2 Table of Errors ............................................................ 119

A.3 Table of Warnings .......................................................... 120
# List of Figures

2.1 A CSmith Generated Program ................................. 6
2.2 CodeSmith Code Example ..................................... 8
2.3 CodeSmith Generator Output Example ....................... 8
2.4 Flow of Machine Learning .................................... 9

3.1 Overview of Genesis ........................................... 13
3.2 Example Instance Programs Generated ....................... 15
3.3 Flow of Genesis ............................................... 17
3.4 Example Instance Programs Generated ....................... 21

5.1 Examples of Instance Programs generated using variables .......... 37
5.2 Genesis Program using `sample` ................................ 38
5.3 Template Program referencing `outputDieRoll` and `outputCoinFlip` .. 38
5.4 Example Instance Programs generated when using `enumerate` ...... 39
5.5 Effect of `enumerate` in Different Sections .................... 40
5.6 Genesis Program using `genif` ................................ 43
5.7 Template Program referencing `loop` .......................... 43
5.8 Example Instance Program generated from `genif` Example ........ 44
5.9 Genesis Program using `genloop` .............................. 45
5.10 Template Program referencing `functionBody` ................ 45
5.11 Example Instance Program generated from `genloop` Example .... 46
5.12 Genesis Program demonstrating effects of the Processing Order . . . . . 47
5.13 Example Instance Program from Processing Order Example . . . . . . 48
5.14 Genesis Program using all sections of Genesis . . . . . . . . . . . . . . 48
5.15 Genesis Flow for this example using all sections . . . . . . . . . . . . 50
5.16 Example Instance Program Set demonstrating each section . . . . . . . 51
5.17 Genesis Program using Multiple enumerate . . . . . . . . . . . . . . . 53
5.18 Genesis Flow of Multiple enumerate Example . . . . . . . . . . . . . . 54
5.19 Example Instance Programs generated with multiple enumerate . . . . 55
5.20 Example C Library File (gen_c.glb) . . . . . . . . . . . . . . . . . 56
5.21 Example of the Use of a Library File . . . . . . . . . . . . . . . . . . . 57

6.1 Image Filtering Genesis Program (IFGP) . . . . . . . . . . . . . . . . . 60
6.2 Template Program for Image Filtering . . . . . . . . . . . . . . . . . . . 61
6.3 Instance Program for Image Filtering . . . . . . . . . . . . . . . . . . . 63
6.4 Template Program for Static Program Characteristics . . . . . . . . . . 64
6.5 Static Program Characteristics Genesis Program (SPCGP) . . . . . . . . 65
6.6 Genesis Output for Static Program Characteristics . . . . . . . . . . . . 66
6.7 Modified Static Program Characteristics Genesis Program (SPCGP2) (Part 1 of 2). The other half of the program is in Figure 6.8. . . . . . . . . . . 67
6.8 Modified Static Program Characteristics Genesis Program (SPCGP2) (Part 2 of 2). The first half of the program is in Figure 6.7. . . . . . . . . . . . 68
6.9 Image Layering Genesis Program (ILGP) . . . . . . . . . . . . . . . . 71
6.10 Genesis Output for Image Layering . . . . . . . . . . . . . . . . . . . 73
6.11 Modified Image Layering Genesis Program (ILGP2) . . . . . . . . . . . 74

8.1 Genesis and Template Programs for Genif Correctness Testing . . . . . . 81
8.2 Genesis Output for Genif Correctness Testing . . . . . . . . . . . . . . 82
8.3 Distribution of Sampling numEpochs . . . . . . . . . . . . . . . . . . . 82
8.4 Chi-Squared P-value Distribution ........................................ 84
8.5 Execution Times of the Preprocessor ................................. 85
Chapter 1

Introduction

There is much interest in *program generators* [9, 12]; i.e., programs that generate other programs [46]. A program generator is typically a compiler that reads a program in a *generator* language, and processes it to create programs in a *target* language. Some program generators in recent years include CSmith [55], TestMake [37], and CodeSmith Generator [14], all of which generate codes in various application domains and target languages.

Large sets of programs are important in various areas of computer science and engineering. In compiler testing, for example, successfully running test programs on a compiler increase the confidence in its functionality and correctness, and assures that it works under different scenarios [42]. Similarly, supervised machine learning (ML) methods for performance auto-tuning rely on large program sets called *training programs*, i.e., programs to which various optimizations are applied and whose characteristics and performance are used to build a ML model. For these models to predict accurately, it is crucial to have a sufficiently large number of training programs where the characteristics are discriminating. That is, the values of each characteristic must be diverse so that these training programs can properly represent the desired program space [39].

However, the number of real programs available for the above purposes is often not
adequate. For compilers, it can be difficult to build up a diverse set of programs that contain enough functionality combinations and error scenarios. Similarly, benchmark suites used to evaluate performance of software and systems [8, 54] usually consist of only tens of programs and are usually too small to build sufficiently large and diverse training sets for ML models. Thus, program generators can be used to generate synthetic programs for use in these situations to compensate.

The use of existing program generators comes with limitations, including a lack of user control over the programs generated [55], inflexible and restrictive use cases and target languages [37, 45, 52], and difficulties using associated tools [37]. Ad-hoc methods of generating large program sets, such as the use of Perl or Python scripts, also have their own limitations. Generating a sufficiently large number of diverse testcases using these scripts is cumbersome and increasingly difficult as the user desires more control over the characteristics of the generated programs. It is also difficult to generate a representative set of programs in a space without exhaustively generating all the programs in that space. Scripts are also difficult to write, maintain, and extend, as the user needs to handle both the expression and generation of programs for each set of desired programs.

In this thesis, we present Genesis, a program generation language to help address the need for simpler synthetic code generation. Genesis simplifies the expression of a set of generated programs and facilitates the generation of synthetic programs in a statistically controlled fashion. Genesis allows a user to annotate a template program to identify code segments of the program they wish to vary. The user also describes the values each parameter in the code segments could take, and the desired statistical distribution of these values in a Genesis program. The Genesis preprocessor uses the annotations in the input template program and the information in the Genesis program to generate a user-specified number of instance programs. The values of each parameter in the generated instance programs is drawn from their corresponding distributions, thus statistically controlling the lengths and characteristics of these programs.
Genesis has many properties that make it a useful tool. It has functionality not present in existing preprocessors or program generators, making Genesis unique in comparison. Genesis provides the useful ability to sample from declared numerical distributions, which in turn generates multiple instance programs that vary based on those sampled values. Genesis can also generate multiple instance programs by enumerating distributions.

Genesis is the first tool for generating multiple synthetic programs that values greater ease while maintaining flexibility at the cost of learning a new language as a trade-off. Genesis removes the need for the user to write code handling the generation of the instance programs, which is handled automatically by the Genesis preprocessor. New programs can be generated by making small changes to the annotations without a need for large overhauls, adding to the ease of use of Genesis. Genesis can also be modular, and code be reused across different Genesis programs.

Genesis is also useful because it is more flexible than other preprocessors, allowing for different ways to describe varying output code snippets. As Genesis is a standalone preprocessor and not an extension a current program language, it is not limited to generating programs for a single output language and can generate programs for multiple different output languages.

We ultimately believe that Genesis is a useful tool that eases the expression of large and diverse program sets, which may lead to better end results for domains such as software compiler testing and program auto-tuning using supervised machine learning.

1.1 Thesis Contribution

This work on Genesis makes the following contributions:

- The creation of the Genesis language. The language is easy to use while providing flexibility to users in how programs are expressed.

- The combination of the synthetic creation of programs with a systematic user-
indicated method of random sampling of values within those programs. Genesis is the first tool that provides these two functionalities together.

- The development of the Genesis preprocessor. The preprocessor generates programs in a reasonable time and samples values that have varying amounts of deviance from their declared distributions, as expected.

1.2 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 gives relevant background material. Chapter 3 gives an overview of Genesis using simple examples that illustrate its use. Chapter 4 details the salient constructs of the language. Chapter 5 illustrates the constructs of Genesis using examples. Chapter 6 describes three case studies that demonstrate how Genesis facilitates the generation of programs in greater detail. Chapter 7 briefly describes our current implementation of a preprocessor prototype. Chapter 8 discusses our evaluation of Genesis. Chapter 9 presents related work. We conclude and present future work in Chapter 10. Finally, Appendix A gives a full specification of Genesis constructs.
Chapter 2

Background

In this chapter, we discuss background material that will be helpful in understanding and motivating the rest of the thesis. Section 2.1 gives background material on program generators. Section 2.2 describes some application domains that can benefit from Genesis.

2.1 Program Generators

As stated in Chapter 1, a program generator is a program that generates other programs [46]. The benefit of using a program generator is that it requires less effort than creating programs by hand. Using a program generator can decrease coding time and remove time spent performing mundane, repetitive tasks [46]. The resultant code is also more consistent to standard patterns and styles and contain fewer bugs [14].

Program generators vary in the amount of user input required. Some program generators require little or no input from the user. Others require the user to write a program in a generator language, which a compiler processes to generate programs in a target language.

In the remainder of this section, we review some of the existing program generators in the literature.
Chapter 2. Background

```c
#include "csmith.h"
static long __undefined;
/* --- FORWARD DECLARATIONS --- */
static int32_t func_1(void);
/* --- FUNCTIONS --- */
static int32_t func_1(void)
{
  int8_t l_2 = 0xC9L;
  return l_2;
}
/* ---------------------------------------- */
int main (int argc, char * argv[])
{
  int print_hash_value = 0;
  if (argc == 2 && strcmp(argv[1], "1") == 0) print_hash_value = 1;
  platform_main_begin();
  crc32_gentab();
  func_1();
  platform_main_end(crc32_context ^ 0xFFFFFFFFUL, print_hash_value);
  return 0;
}
```

Figure 2.1: A CSmith Generated Program

2.1.1 CSmith

CSmith [55] is an example of a program generator that requires minimal effort and input from the user. It is a randomized test-case generator that generates programs in C or C++ for use in detecting bugs in their respective compilers. These programs, after compiling, perform random computations and prints a final result to the terminal. CSmith also performs the same computations to calculate a result. A bug in the compiler is found if both results differ, or if the compiler crashes. The simplest example of a program that CSmith generates is shown in Figure 2.1. Programs generated by CSmith can range from tens of lines to possibly thousands of lines long.

CSmith allows some diversity for the programs it generates based on pre-set properties. Some properties, such as the use of data types, computation types, or checksum calculations, can be enabled or disabled. Other properties control the maximum values of program characteristics, such as for the number of array dimensions, fields in a structure, or pointer depth. Otherwise, there is little user control over the programs generated,
restricting the possible uses of CSmith. It is further restricted by the limited languages of
the programs it generates. The lack of user input and the randomness of the program it
generates also make the programs difficult to read and understand.

2.1.2 TestMake

Another example of a program generator that requires minimal input from the user is
TestMake [37]. It generates test harnesses around a user-indicated piece of code. TestMake
requires the use of an external toolset, called the McCabe Toolset, which analyses the
indicated piece of code, and reports on the test-paths and data usage of the code. Using
this information, TestMake generates harnesses with inputs causing the code to enter
different code-paths.

The language of choice for TestMake is Fortran 90, but it works with other languages
like C and Java. Similar to CSmith, limitations restrict its possible uses. The user has
little control over the programs generated. TestMake also works best for small code
fragments and code with fewer state variables. These limitations can only be overcome
by defining new testcases manually, defeating the purpose of a program generator.

2.1.3 CodeSmith Generator

An example of a program generator that allows more user input is CodeSmith Generator
[14]. It generates source code by using templates containing CodeSmith Generator
code. A compiler detects, processes, and modifies the code using information stored in
database tables. This generally is used to minimize repetitive tasks, such as outputting
values in a long database table.

An example of a template using a CodeSmith loop to repeatedly access a table is
shown in Figure 2.2. This piece of code lists out the names and ages of students under the
CHAPTER 2. BACKGROUND

2 respective headings. CodeSmith Generator code is placed between angular brackets on lines 3-5. In this code, the names and ages of students are extracted from a database table, which possibly looks like Table 2.1. After being processed by the compiler, the entire range of CodeSmith Generator code is removed and replaced by the result of running the code, as shown in Figure 2.3. Thus, the repetitive, manual task of listing out all the students by hand is eliminated using CodeSmith Generator.

A limitation of CodeSmith Generator is that it is not suited for generating large sets of programs, as these templates must be written by the user one by one. While this example was simple, another limitation is that current program generator languages become complicated to learn as the number of features in the language increases, making some languages difficult to learn for specific domains and situations.
2.2 Application Domains

2.2.1 Machine Learning

Machine learning (ML) involves the building of systems that can learn from data. A common method in machine learning is supervised learning, illustrated in Figure 2.4. In this method, a set of past training data, consisting of input values and corresponding output values, is used to build a ML model during the first phase of training. This ML model can then be used to predict the unknown output of unseen input data in the second phase of testing. Examples of ML models for supervised learning include support vector machines, decision trees and random forests [39].

ML methods can be used to improve program performance, particularly on parallel architectures such as multi-cores and Graphics Processing Units (GPUs) [7, 15, 23, 26, 49, 53]. Much of the previous work in this field uses supervised learning methods that rely on training programs and program optimization, i.e., programs to which various optimizations are applied and whose characteristics and performance are used to build a
machine learning model. The input values to the model are *program features* or *program characteristics*, such as loop characteristics or memory access patterns, which are extracted from the input training programs. The output value is a measure of the performance of the program. The goal of building the ML model is to use it to predict if a given program optimization will be beneficial to the performance of programs given extracted input characteristics.

The diversity and quality of the input training programs used to gather input data from is of vital importance [20]. A good quality, diverse set of input training data results in a better model and more accurate predictions. A poor set results in a model that predicts inaccurately. The quality of the learning achieved by these ML models is often a function of the degrees of balance and overlap of the training programs [5]. It is crucial to use characteristics that are discriminating or different from one another and to have a sufficiently large number of training programs in which the values of each characteristic are diverse [39]. Having a sufficiently large amount of diverse training programs is beneficial for the ML model to have an accurate view of the program space and predict more accurately.

Traditionally, the input training programs consist of existing benchmarks. One limitation of existing benchmarks are that the number of programs available for training is small. Benchmark suites commonly used to evaluate performance of software and systems, such as PARSEC [8] or SPLASH-2 [54], usually consist of only tens of programs. Such a training set is usually not sufficiently large or diverse enough to build a ML model that spans the desired program characteristics.

One potential alternative is to use synthetic programs generated as needed for the training [26]. These programs can be used as a training sample for a ML model. Previous work shows their value in this context [26, 28]. These generated training programs must be able to express the large variation of possible inputs to the ML model, spanning and satisfying the necessary characteristics and features. However, generating a sufficiently
large number of synthetic programs is cumbersome and it is often unclear if the values of a characteristic in the programs are sufficiently diverse or if they span the desired program space. Thus, a need exists for a more efficient way to generate these synthetic training sets to fill the gap in previous literature.

### 2.2.2 Compiler Testing

Testing is a fundamentally important aspect of all areas of computer science and engineering [42]. Compilers are one of the most important grand challenges in computer science [29]. Adequate compiler testing allows compiler programmers to ensure their code can handle a large set of possible inputs, increase confidence in the functionality of their programs, and assure that the code works under different CPU loads, stresses, and operational capacities [42]. Poor testing can contribute to compiler releases with buggy code, poor performance, or security exploits resulting in the loss or release of insecure data. Thus, testing is important for compilers before its release and throughout its lifetime.

To properly perform adequate testing on compilers, a large amount of input test programs is required. Creating these test programs can take a large amount of time and resources to create various possible cases and interactions between components. When not enough real input test programs exist for these compilers, synthetic test programs can be used to compensate. These generated test programs should contain code that tests the basic functionality of the compiler for general cases. In addition, it is also important to have a mix of different functionalities together to properly test corner cases that occur less often, but with similar consequences.

Testcases for compilers and software can also be helpful in other situations. For example, a large amount of testcases can be useful in testing during a school programming class. A large amount of synthetically-generated testcases can compensate for not having a large amount of user-created testcases, increasing the ease for testing and marking, while decreasing the work required.
Chapter 3

Genesis Overview

Genesis defines a language whose constructs make it easy to generate synthetic programs. A user of Genesis starts with a template program in a target language of choice (C or C++, for example). The user identifies segments of this program that are to vary across a set of different instance programs, and replaces these segments with constructs of the Genesis language. The user also writes a Genesis program expressed in the Genesis language that describes how these segments are to be varied. The user then runs the Genesis preprocessor to process the Genesis and template programs, using these programs to generate a set of instance programs. These instance programs follow the form of the same template program, but are varied in the segments identified by the user. This high-level flow is depicted in Figure 3.1.

A simple example can be used to illustrate this flow and demonstrate some of the constructs of Genesis. Consider the following program, expressed in the C++ language:
In this program, the value of \(x\) is read from the standard input. This value is multiplied by 2 and the result is printed to the standard output. For this example, it is desired to generate a set of instance programs in which the integer multiplier of \(x\) is not always 2, but randomly varies from 1 to 10. Genesis can be used for this purpose.

The user of Genesis modifies the code above into a template program. A Genesis feature reference \${\text{multiplyTerm}}\) replaces the term \(2\times x\), which is the part of the code which the user wishes to vary, as shown below.
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    cout << ${multiplyTerm} << endl;
    return (0);
}

This feature reference refers to a Genesis feature `multiplyTerm` in the Genesis program, which the user writes to define the code snippet and how it should vary. Thus, the user also writes a Genesis program containing this feature, as shown below.

```plaintext
1 begin genesis
2   distribution range = {1:10}
3
4 feature multiplyTerm
5   value coefficient sample range
6     ${coefficient}*x
7 end
8
9 generate 4
10
11 end genesis
```

The Genesis program uses the `begin genesis` and `end genesis` constructs on lines 1 and 11, respectively, to delimit Genesis code. Line 2 defines a Genesis `distribution` named `range`, which indicates the range of values from which the multiplier is to be randomly chosen, or `sampled`. The `feature definition` on lines 4 to 7 defines the `multiplyTerm` feature. It contains the declaration of a Genesis `value` named `coefficient` whose value will be randomly sampled from the distribution `range`. The code snippet of the multiply term itself is on line 6. This line uses the sampled value to `coefficient` using a `value reference`. The sampled value of `coefficient` will replace the reference, producing a final code snippet. Finally, line 9 uses the `generate` construct to indicate that 4 instances programs are to be generated.

The Genesis program and template program are then read by the Genesis preprocessor. It creates a copy of the template program to start generating an instance program. It scans
the instance program for feature references and for each such reference, the corresponding feature defined in the Genesis program is *processed*. The preprocessor processes a feature by evaluating the Genesis code in its feature definition and producing a resultant code snippet, called a *feature instance*. This feature instance *replaces* the feature reference in the template program. In this example, \( \{ \text{multiplyTerm} \} \) is replaced by the code snippet returned by evaluating its feature definition. This produces a newly generated instance program with no remaining feature references.

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    cout << 5*x << endl;
    return (0);
}
```

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    cout << 8*x << endl;
    return (0);
}
```

*Figure 3.2: Example Instance Programs Generated*

This process repeats multiple times to generate multiple instance programs. Each instance program takes on the form of the template program, with a new processing of \( \text{multiplyTerm} \) replacing its reference each time. In each processing of \( \text{multiplyTerm} \), the value \( \text{coefficient} \) is sampled again, taking on a new value ranging from 1 to 10. The result is a set of 4 instance programs that differ based on this sampling of \( \text{coefficient} \).

*Figure 3.2 shows two possible generated instance programs.*

Although this example is in C++, Genesis can work with different output languages, such as C (and its macro languages), Java and machine language. To keep Genesis language agnostic, it is up to the user to ensure that the instance programs are both correct syntactically (such that the instance program is valid) and semantically (such that the instance program runs correctly) in the target language.
3.1 The Flow of Genesis

As stated earlier, the preprocessor takes two inputs: a template program, expressed in a standard programming language, such as C, Java, or C++, and a Genesis program, expressed using the Genesis language.

The template program is a piece of code written in the standard programming language that each generated instance program will take its form of. The template program contains references to Genesis features to indicate the location of their intended use. These Genesis features are defined by the user in the Genesis program. The Genesis program defines Genesis entities, such as distributions, values, and features, also using code in the standard programming language mixed with Genesis code. The code in the standard programming language are written with references to other Genesis entities to indicate the locations of their use. These references are in the form of their given Genesis name, a user-given name to an entity, wrapped between $\{}$ and $\}$. For each instance program generated, value references will be replaced with their sampled values and feature references will be replaced by a processed feature instance. All feature references in the template program will be replaced by a differently processed feature instance, resulting in multiple varied instance programs.

Both the template program and features can contain references. The template program can only contain feature references. In contrast, feature definitions can contain feature references, as well as references to Genesis values and other Genesis entities. Genesis entities sample values at the point of their declaration and references to Genesis entities can only be used if that entity has already sampled a value. The exception to this are feature references, which processes a new feature instance once referenced.

Figure 3.3 depicts the flow of processing in Genesis. The preprocessor first reads the Genesis program to store its information. A Genesis program contains 3 sections: the global section, the program section, and the feature definition section. The preprocessor parses and stores all features that were defined in the feature definition section. This
section contains the definitions of all features. Feature definitions are given a Genesis name and are processed once each time the feature is referenced.

Second, as depicted in the figure in the box labelled “Process Global Section”, the preprocessor processes the global section of the Genesis program, processing each statement sequentially. The global and program sections both contain Genesis values that define and perform samplings from distributions. The global section contains Genesis entities which are sampled once for the entire set of generated instance programs. These sampled values are stored for use in all instance programs.

Next, as depicted in the box labelled “Process Program Section”, the preprocessor processes the program section sequentially, where the values defined in this section are sampled. The preprocessor reprocesses the program section once for every instance program. These sampled values are stored for use only for that instance program, with each entity sampling new values for each instance program.
Thus, the two sections differ by having different processing rates and by storing sampled values for different lengths of time. This controls the number of programs a sampled value should be held constant for. The entities defined in these two sections can be referenced in any feature. However, entities defined within a feature exist only in that feature and cannot be used outside that feature. Genesis entities with the same Genesis name can be defined in different areas of the Genesis program, but two Genesis entities with the same name cannot exist with a sampled value at the same point of time. An example demonstrating different processing rates is shown in Section 5.6.

Then, as depicted in the box labelled “Process Template Program”, the preprocessor then begins to generate an instance program by copying the template program. The preprocessor then searches for any feature references in the template program and, if any are found, processes the corresponding feature definition. The preprocessor replaces Genesis references in the definition by their sampled values or processed feature instance. This produces an actual code snippet that replaces the feature reference in the template program. A feature is generally processed once each time the feature appears in the template program. This processing repeats for every feature reference, creating a final instance program in the target language with no more feature references.

The preprocessor repeats the last two steps repeatedly to generate the indicated number of instance programs written in the `generate` statement. Each instance program at first is a copy of the template program. For each copy, the preprocessor re-samples the values in the program section, and processes the template program again with referenced features reprocessed. Processing all the copies results in the final, generated set of instance programs.
3.2 A Second Example

In this section, another simple example is used to reinforce the constructs of the Genesis language and the processing of a Genesis program. For this example, it is desired to generate code that contain two memory accesses inside a single loop as shown:

```cpp
for (int i = 0; i < n ; ++i) {
    t1 = x[c1*i+s1];
    t2 = x[c2*i+s2];
}
```

In this example, two reads are made to an array, \( x \), in each iteration of a loop. The coefficients, \( c1 \) and \( c2 \), and the offsets, \( s1 \) and \( s2 \), determine the sequence of memory locations accessed and thus are likely to affect the memory performance of an underlying architecture. Genesis can be used to generate multiple programs in this form that have different values of these four constants. For this example, \( c1 \) and \( c2 \) should be uniformly distributed over the range 1 to 4 and that \( s1 \) and \( s2 \) should be uniformly distributed over the range 0 to 7.

To generate multiple programs containing this loop structure, the code is first converted into a template program. The template program is essentially the loop code as shown previously, but with the memory accesses replaced by references to the feature `mem_access`, as shown below.

```cpp
for (int i = 0; i < n ; ++i) {
    t1 = ${mem_access};
    t2 = ${mem_access};
}
```

The feature `mem_access` is defined in the Genesis program shown below.
begin genesis
  global
distribution coef_dist = {1-4}
distribution offs_dist = {0-7}
end

feature mem_access
value coef sample coef_dist
value offs sample offs_dist
  x[coef*i + offs]
end

generate 15
end genesis

The feature mem_access is defined on lines 7 to 11. The memory access is placed on line 10 as the code snippet $x[\text{coef}*i + \text{offs}]$. The two Genesis names coef and offs are used in this code snippet. The values of coef and offs will be sampled from the distributions coef_dist and offs_dist as indicated by the sample constructs in their definitions. These distributions are defined using the distribution construct in the global section, respectively with their desired ranges.

The Genesis program is read first and parsed by the preprocessor. The information is stored before the preprocessor begins reading the template program. The generate statement on line 13 of the Genesis program instructs the preprocessor to generate 15 instance programs, each originally a copy of the template program. In each instance program, the preprocessor processes the feature mem_access twice, sampling the values of each Genesis entity from its respective distribution, and using the processed feature to replace each feature reference. Each processed feature contains a different sampling of the Genesis entities in the feature, resulting in a total of 15 different instance programs, each with two differently processed mem_access feature instances with differently sampled coef and offs. Figure 3.4 shows some examples of the programs produced.
Figure 3.4: Example Instance Programs Generated

```java
for (int i = 0; i < n; ++i) {
    t1 = x[1*i + 2];
    t2 = x[1*i + 7];
}
```

```java
for (int i = 0; i < n; ++i) {
    t1 = x[2*i + 5];
    t2 = x[3*i + 4];
}
```
Chapter 4

Genesis Constructs

Genesis provides several constructs for describing instance programs and how the code should vary between them. In this section, we give an overview of the more salient ones. A full description of the constructs appears in Appendix A.

Genesis constructs are designed to describe different code types, while keeping the Genesis program readable to the user. Using a construct tells the preprocessor to interpret that line as a Genesis line, and not to output it to the instance programs. A backslash can be used as an escape character in front of a Genesis construct to output a Genesis line to the instance programs.

For simplicity, lines in code snippets beginning with `print` are generic print statements in some target language, and are not specific to Genesis.

4.1 The `distribution` Construct

The `distribution` construct specifies a set of values and their corresponding probabilities. For example,

\[
\text{distribution a_dist} = \{1\{0.7\}; 2\{0.1\}; 4\{0.1\}; 8\{0.1\}\}
\]

defines a distribution named `a_dist` with values: 1, 2, 4, and 8, each with the probability shown in the curly braces next to it. If the probabilities are omitted, a uniform distribution
It is also possible to define uniformly distributed ranges of values. For example,

\[
\text{distribution s_dist} = \{1:10\}
\]

defines a distribution named \textit{s_dist} where all integer values from 1 to 10 can be sampled with equal probabilities.

Currently, Genesis only allows uniform distributions in this form, but other distribution types (e.g. normal distribution) can possibly be implemented in the future. By default distributions can contain integers and strings. It is possible to use the modifier \texttt{real} to allow real distributions, as in the following example:

\[
\text{distribution s_dist} = \{1:10;;\text{real}\}
\]

This distribution, \textit{s_dist}, allows real values between 1 and 10.

Distribution can be defined using Genesis values. For example,

\[
\text{value upperBound sample } \{5:10\}
\text{distribution b_dist} = \{1:$\{\text{upperBound}\}\}\}
\]

defines a distribution named \textit{b_dist} created with a bound using \textit{upperBound}, a previously sampled Genesis value.

Distributions are set once their definition line is processed, and do not change during processing. Distributions defined using Genesis values do not change if the Genesis value changes later on in processing. An example demonstrating this is shown in Section 5.5.

### 4.2 The value Construct

The \texttt{value} construct defines a Genesis entity whose value is sampled from a distribution. Values can be propagated as constants to the instance programs or can be used in the definition of other constructs. An example of a value line is:

\[
\text{value stride sample s_dist}
\]

This declares the Genesis entity with the given Genesis name \textit{stride}, whose value is sampled from the distribution \textit{s_dist} using the \texttt{sample} construct. Thus, assuming the
s_dist defined above, stride equally likely takes a value from 1 to 10 each time it is sampled. A reference to stride in a feature is replaced by the sampled value when the feature is processed.

The assigned distribution can be defined in-line:

\[\text{value stride sample } \{1:10\}\]

This distribution is functionally the same as the previous example. This allows for a simpler declaration, but removes the ability for reuse of the same distribution.

Instead of sampling from a distribution, a Genesis value can also enumerate one. Thus,

\[\text{value stride2 enumerate s_dist}\]

makes stride2 take on every possible value of s_dist, one per instance program. Many examples using enumeration are in Chapter 5.

Values defined over multiple lines can be compressed to a single line. For example,

\[
\begin{align*}
\text{value stride1 sample a_dist} \\
\text{value stride2 sample a_dist} \\
\text{value stride3 sample s_dist} \\
\text{value stride4 sample s_dist}
\end{align*}
\]

can be simplified to

\[\text{value stride1,stride2 sample a_dist; stride3,stride4 sample s_dist}\]

Lastly, values can be set without sampling from a distribution. For example,

\[\text{value stride=2}\]

declares the Genesis entity named stride, and sets its value to 2, equivalent to a Genesis value sampling from a distribution containing only the value 2. This value also can be referenced in the same way using its Genesis name.

### 4.3 The varlist Construct

The varlist construct defines a pool of variables for use in a processed feature and hence, be part of the instance program. Along with the varlist construct, the created pool of variables itself is also called a varlist. A varlist is analogous to a distribution as entities
which can be sampled from. A example of a varlist line is:

```
varlist my_vars[5]
```

This defines a pool named `my_vars` of size 5. Five variables in the target language, named `my_vars1` to `my_vars5`, can be sampled from this varlist using Genesis variables, described in Section 4.4.

The names of the variables in the varlist can be changed using a `name` modifier, as shown:

```
varlist my_vars[5] name(temp)
```

The given Genesis name of the varlist remains `my_vars`, and this Genesis name is used to refer to this varlist. It contains 5 variables ranging from `temp1` to `temp5`.

If a varlist was manipulated using `add` or `remove` constructs (described in Section 4.8), a varlist is `reinitialized` by undoing those actions, returning to a full varlist with all its variables. The section in which the varlist is declared indicates the reinitialization rate of the varlist. Varlists declared in the global section are created once for the entire set of programs. The size of the varlist and state of variables are maintained between instance programs in this case. Varlists declared in the program section are reinitialized at the point of its declaration, and thus, it return to a full varlist with all its variables for each instance program. Varlists declared in a feature are local to that feature only and are reinitialized for each processing of the feature.

It is possible to create a pool of variables using an existing varlist. For example,

```
varlist other_vars from my_vars
```

defines another pool of variables named `other_vars` containing all the variables in the `my_vars` varlist. This allows manipulation of two separate varlists with the same set of `my_vars` variables.

Varlists can be referenced with an argument to query information from the varlist. This includes the size of the varlist (using `(size)`), the name used for the variables in the varlist (using `(name)`), and a specific variable name for a variable in a varlist (using a
number). For examples,

```java
value stride1 sample a_dist
varlist my_vars[5] name(foo)
print "$\{\text{temp(size)}\}\$";
print "$\{\text{temp(name)}\}\$";
print "$\{\text{temp(4)}\}\$";
```

The first varlist reference outputs 5, the varlist’s size. The second reference outputs `foo`, the name used in all the variables in the varlist. The third reference outputs `foo4`, the specific name of the 4th variable in the varlist.

### 4.4 The variable Construct

The **variable** construct, analogous to the **value** construct, defines a Genesis entity whose value is sampled from a varlist and is propagated as a variable name to the target program. For example,

```java
variable dest from my_vars
```

defines a Genesis entity named `dest`. Its value is sampled from the varlist named `my_vars`, previously defined. For each sample, the variable used in the instance program is a variable from `my_vars1` to `my_vars5`. An occurrence of `$\{\text{dest}\}$` in a feature is replaced by this variable name when the feature is processed.

### 4.5 The feature Construct

The **feature** construct defines a code snippet built up using Genesis names or possibly other features.

#### 4.5.1 The feature Definition

A feature is defined inside its own block in the feature section of the Genesis program. For example,
feature computation
  variable dest, src1, src2 from my_vars
  ${dest} = ${src1} * ${src2};
end

defines a feature named computation that has the code snippet ${dest} = ${src1} * ${src2};. The variable construct defines three Genesis variables sampled from my_vars. Thus, each time the feature computation is processed, the variables dest, src1 and src2 are sampled to select three variables from my_vars1 to my_vars5. The sampled values replace the corresponding variable references in the code snippet.

A code snippet spanning multiple lines returned by a feature can be condensed to a single line using a singleline modifier before the name of the feature.

4.5.2 feature Usage

A feature is used in the template program or in another feature with a reference, consisting of its name wrapped between “$” and “}”. A feature is processed on demand for each feature reference. The resultant feature instance is substituted into that feature reference only, and each feature reference is substituted by a newly generated feature instance. For example, using the above definition of computation, when ${computation} is detected in the template program, the definition is processed at that point. The feature reference is replaced by ${dest} = ${src1} * ${src2};, with each variable reference replaced by a previously sampled variable.

4.5.3 Multiple feature References

Multiple references to the same feature can be compacted by using square brackets. For example,

${computation[5]}

processes computation five times and replaces this reference with the five instances.

A previously sampled Genesis value can be used instead of an integer. For example,
samples `num_comps` and then `computation` is processed `num_comps` times consecutively.

### 4.5.4 feature Arguments

Features can also have arguments passed in by value. For example,

```plaintext
feature accessWithArg(offset)
    my_vars1 = arr[${offset}];
end
```

defines a feature called `accessWithArg`, where `offset` is passed in, and its value is substituted into the code snippet in the same manner as another Genesis entity. In this case, the feature reference must include an argument. For example,

```plaintext
value passedValue sample {1:3}
${accessWithArg(${passedValue})}
${accessWithArg(5)}
```

shows two references to `accessWithArg`. The first reference uses the Genesis value `passedValue`, taking on a sampled value from 1 to 3. Alternative, a string constant can be used instead; the second reference shows this by passing in `5` to the feature.

### 4.5.5 Stored features

Processed features can also be stored for later referencing by a given Genesis name. A reference with the given Genesis name is substituted by the already processed code without reprocessing, similar to a Genesis value or variable. Consider the following code snippet:
The first line processes an `access`, and stores the code snipped in `stored_access`. Thus, when processing reaches the first `stored_access` reference, the code snippet previously processed replaces it. Both `stored_access` references are replaced by identical code snippets, without any further sampling of its values and variables. In contrast, the `access` feature reference used afterwards still work as described before. This reference is processed during the time the reference is detected, and substituted into that feature reference.

In contrast to a regular feature reference, which provides quick process-and-replace functionality, using stored features allows a user to separate these two functions of processing and replacement, allowing multiple replacements as necessary for a single processing of a feature.

### 4.6 The `generate` Construct

The `generate` construct defines the number of instance programs to generate. For example,

```
generate 5
```

indicates that we wish to generate 5 instance programs.

The `generate` construct allows the definition of global distributions.
generate 5 with
   a_dist={1{0.7};2{0.1};4{0.1};8{0.1}}
   b_dist={1:6}
end

Also, it is possible to have multiple `generate` constructs with different global distributions. For example,

```plaintext
generate 10 with a_dist={1:5}
generate 5 with a_dist={16:20}
```

generates fifteen total instance programs. It defines `a_dist` as a distribution from 1 to 5 for ten instance programs, and as a distribution from 16 to 20 for five instance programs.

### 4.7 The `genmath` Construct

The `genmath` construct allows the updating of previously sampled values. Consider the following example:

```plaintext
value testValue sample {1:5}
print "Current value is \$\{testValue\}";
genmath testValue = \$\{testValue\}+5
print "Current value is \$\{testValue\}";
```

This samples a `testValue` value from 1 to 5. After replacing the value in the following line, it increases the value of `testValue` by 5. This results in the last line having a different replaced `testValue` value.

The `genmath` statement is flexible enough for other uses. Genesis values can be used as counters with the `genmath` statement. An example of using `genmath` statement with other Genesis statements for the conditional generation of code is shown in Section 4.10.

### 4.8 The `add` and `remove` Constructs

The `add` and `remove` constructs modify a varlist to affect future samplings. For example,
variable dest1, src1, src2 from my_vars
${dest1} = {src1} * {src2};
remove dest1 from my_vars
variable dest2 from my_vars
${dest2} = {src1} * {src2};
add dest1 to my_vars

prevents dest1 and dest2 from sampling the same variable by removing the sampled variable of dest1 from the my_vars varlist before dest2 is sampled. The add re-adds the sampled variable of dest1 back to the my_vars varlist to allow it to be selected by future samplings.

Ultimately, it is the obligation of the user to write Genesis programs to control the desired code in the instance program. These two constructs gives the user the flexibility to control code as desired. For example, these constructs can be used to generate code that will not be detected by a later compiler stage as dead code.

### 4.9 The genif Construct

The genif construct is used for conditional generation of code snippets. Consider the following example:

```gen
value conditionValue sample {1:3}
genif ${conditionValue}==1
    ${computation}
end
```

The above code samples a value from 1 to 3 for conditionValue. If the value sampled is 1, then computation is processed and placed into the instance program. Otherwise, this section of the Genesis program is processed, but produces no code as a result. This means that this code results in a computation one third of the times this code is processed. The genif construct does not generate if statements in the instance program, and is only used to control the flow through the preprocessor.

Using genelsif constructs after a genif statement allow for a second condition block that is only evaluated if the first genif statement is evaluated to false. Also, genelse
constructs allow a code section to be processed if all preceding `genif` and `genelsif` statements were evaluated to be false. For example,

```plaintext
value conditionValue sample {1:3}
genif ${conditionValue}==1
   ${computation}
genelsif ${conditionValue}==2
   ${memory_access}
genelse
   ${memory_load}
end
```

shows a code snippet illustrating all three conditional constructs in Genesis. The Genesis value `conditionValue` samples a number from 1 to 3. If 1 is sampled, a `computation` is generated and replaces the reference. If 2 is sampled, a `memory_access` is generated and replaced. Else, a `memory_load` is generated. Thus, assuming all three features produce a resultant code snippet, this example generates a code snippet every time it is processed.

### 4.10 The genloop Construct

The `genloop` construct facilitates repetitive code generation. A genloop iterator is created as part of the `genloop`. Consider the following example:

```plaintext
genloop loopVar:1:5
   ${access(${loopVar})}
end
```

This `genloop` construct creates a genloop variable called `loopVar` which iterates from 1 to 5. The example produces five references to the feature `access`, each with a different value from 1 to 5 passed in as an argument. Similar to the `genif` construct, this construct does not produce a loop in the instance program, but instead 5 consecutive versions of the code produced when the `access` feature is processed, each replaced by a new processing of the feature when the code is processed.

The `genloop` construct can also test boolean conditions, similar to a C while loop. Consider the following example:
genloop ${testValue} < 5
    genmath testValue = ${testValue}+1
    print "Current value is ${testValue}";
end

This repeatedly generate code snippets that reference testValue. During each iteration, testValue increases its value by one. This code stops processing when testValue becomes greater than 5 and the boolean condition is checked at the beginning of the genloop iteration.

### 4.11 The geninclude Construct

The geninclude construct allows Genesis code in another file to be used in the current Genesis program. Usually, this construct is used with pre-made library files provided by Genesis, which implement useful feature definitions that may be useful across multiple Genesis programs of the same target language. Library files are explained in greater detail in Section 5.9. It can also be used to split the Genesis program into multiple files.

For example,

```
geninclude gen_c.glb
```

makes available those features defined in gen_c.glb, a library containing features that declare and initialize variables in C programs. One example of these features is varlistDeclare, which initializes C variables in an indicated varlist.

### 4.12 The genassert Construct

The genassert construct makes an assertion of a boolean expression, similar to a genif statement. If the expression is evaluated to be true, processing of the instance program continues normally. However, if it is false, processing stops for the current instance program and the program is deleted. The preprocessor then continues processing the next instance program. For example,
defines two Genesis values, sampled from 1 to 5. The `genassert` construct calculates the product and asserts that it is not 1 (i.e., 1 is not sampled for both values). If the product is 1, the generation of that instance program is aborted, and that program is not included in the final set of instance programs.
Chapter 5

Genesis Examples

In this chapter, we provide examples of Genesis constructs defined in Chapter 4. These examples also demonstrate how some of the constructs interact.

5.1 Using Variables

The first simple example demonstrates in greater detail the use of the variable construct and how its use differs from the value construct. For this example, the code example given in Section 3.2 is used:

```c
for (int i = 0; i < n ; ++i) {
    t1 = x[c1*i+s1];
    t2 = x[c2*i+s2];
}
```

This program performs two reads to an array, x, in each iteration of a loop, as before. The coefficients, c1 and c2, and the offsets, s1 and s2, are likely to affect the memory performance of the underlying architecture. It is desired to generate multiple programs that have different values of the four constants using Genesis. To extend this example, it is now also desired to vary t1 and t2, their values ranging from temp1 to temp5 randomly.
The template program shown below is modified slightly from the template program shown in Section 3.2.

```c
for (int i = 0; i < n ; ++i) {
  :
  ${mem_access};
  :
  ${mem_access};
  :

```

Now, \( t1 \) and \( t2 \) will also be sampled along with the coefficients and offsets. Thus, they are removed from the template program and are added to the `mem_access` definition, defined in the Genesis Program below.

```c
1 begin genesis
2   global
3     distribution coef_dist = {1-4}
4     distribution offs_dist = {0-7}
5     varlist dest_list[5] name(temp)
6   end
7
8   feature mem_access
9     variable dest from dest_list
10    value coef sample coef_dist
11    value offs sample offs_dist
12    ${dest} = x[${coef}*i + ${offs}]`
13 end
14
15 generate 15
16
17 end genesis
```

The Genesis program has changed to include a `varlist` line and a `variable` line. The `varlist dest_list` is included in the `mem_access` feature on line 5, initialized to a size of 5. Thus, this varlist contains 5 variables, ranging from `temp1` to `temp5`. The `variable dest` samples a variable from the `dest_list` on line 9, similar to a value sampling from a distribution. The sampled variable is used on line 12, where `dest` is referenced in the same way as `coef` and `offs`.

When the Genesis program and template program are read through the preprocessor,
15 instance programs are generated. Figure 5.1 shows some examples of the programs produced, with dest replaced by the sampled variable.

```
for (int i = 0; i < n; ++i) {
    temp2 = x[1*i + 2];
    temp5 = x[1*i + 7];
}
```

```
for (int i = 0; i < n; ++i) {
    temp3 = x[2*i + 5];
    temp1 = x[3*i + 4];
}
```

Figure 5.1: Examples of Instance Programs generated using variables

### 5.2 Using enumerate

The enumerate construct breaks away from the Genesis notion of random sampling by allowing a Genesis value to take on each value in a distribution exactly once, one per instance program. This example demonstrates in greater detail the functionality of a single enumerate construct. For this example, we consider both a die roll and a coin flip. Multiple programs are desired where the resulting two probabilistic events are hard-coded but are different for each program. Using Genesis, the die roll is sampled from a distribution of 1 to 6, and the coin flip is from a distribution containing “heads” and “tails” using the Genesis lines as follows:

```
program
    value dieRoll sample {1:6}
    value coinFlip sample {heads, tails}
end
```

Because these Genesis values are in the program section, new values are sampled for these Genesis entities for each program generated. Two features, outputDieRoll and outputCoinFlip, are defined using these Genesis entities:
begin genesis

program
  value dieRoll sample {1:6}
  value coinFlip sample {heads,tails}
end

feature outputDieRoll
  print "Die roll: ${dieRoll}!"
end

feature outputCoinFlip
  print "Coin flip: ${coinFlip}!"
end

generate 3
end genesis

Figure 5.2: Genesis Program using sample

#include <iostream>
using namespace std;

int main() {
  ...
  ${outputDieRoll}
  ${outputCoinFlip}
  return (0);
  ...
}

Figure 5.3: Template Program referencing outputDieRoll and outputCoinFlip

feature outputDieRoll
  print "Die roll: ${dieRoll}!"
end

feature outputCoinFlip
  print "Coin flip: ${coinFlip}!"
end

These features output the result of the two Genesis values sampled in the program section.
The program section and these two features are joined together into a Genesis program, shown in Figure 5.2, with a generate 3 statement.

A template program with references to these features is shown in Figure 5.3. The Genesis program and the template program together generate 3 programs, with a randomly sampled die roll and a randomly sampled coin flip hard-coded in each program.

Now we assume instead, for each random die roll, two programs are to be generated. Both programs will use the same die roll result, but one program will contain the coin
include <iostream>
using namespace std;
int main () {
    ...
    print "Die roll: 3!"
    print "Coin flip: heads!"
    return (0);
    ...
}

(a) Program 1 in Program Set

include <iostream>
using namespace std;
int main () {
    ...
    print "Die roll: 3!"
    print "Coin flip: tails!"
    return (0);
    ...
}

(b) Program 2 in Program Set

Figure 5.4: Example Instance Programs generated when using enumerate

flip result as heads, and one with the coin flip result as tails, thus using both possible
coin flip results. This is accomplished with the enumerate construct. To achieve this, the
program section is changed as follows:

    program
        value dieRoll sample {1:6}
        value coinFlip enumerate {heads, tails}
    end

When the global section is processed, the lines are processed sequentially. The value
dieRoll is processed first and given a sampled value. Once the preprocessor reaches the
enumerate construct for coinFlip, it keeps the previously sampled value for dieRoll,
and generate two instance programs, one with coinFlip taking on the value “heads”,
and one with coinFlip taking on the value “tails”. Each instance program continues
processing independently. Possible examples of the first two programs generated are
shown in Figure 5.4.

The generate 3 in the Genesis program normally results in 3 programs being gener-
ated. However, since the value coinFlip uses the enumerate construct in the program
section, the meaning of generate 3 changes from generating 3 programs to generating 3
enumerated sets. Since the distribution used by coinFlip contains 2 values to enumerate
through, each enumerated set contains 2 instance programs, resulting in 6 total instance
programs generated.

When enumerate is used in the Genesis program, the generate number has a different
Figure 5.5: Effect of `enumerate` in Different Sections
meaning depending on the location of the enumerated value. Enumerated values can be placed in either the global section or the program section, both of which affect the flow of Genesis differently. Figure 5.5 illustrates the difference between the two using a Genesis value enumerating through 3 values and a \texttt{generate 3} statement. When a value being enumerated is in the global section, as shown Figure 5.5a, the preprocessor first processes the value using \texttt{enumerate} before the \texttt{generate} construct, and the entity takes on all 3 possible values. When the global section is finished processing, the preprocessor reads the \texttt{generate} construct with each of the possible enumerated values. The preprocessor generates 3 programs with each possible value, creating a total of 9 programs. In this case, the preprocessor generates 9 total sets of programs, with each set having 1 instance program and with each enumerated value creating 3 sets.

If a value is being enumerated in the program section, as shown in Figure 5.5b, the preprocessor processes the \texttt{generate} construct first, and the number in the \texttt{generate} statement determines the number of instance program sets to generate. For each instance program, the preprocessor processes the program section once, and thus, when the preprocessor processes the enumerated value for each instance program, it turns that instance program into a program set. Each program set contains a program for the 3 possible values in the enumeration, resulting in 3 total sets of 3 instance programs each.

Thus, the total number of programs generated is:

\[ N = E_G \times G \times E_P \]

where \( N \) is the total number of programs generated, \( E_G \) is the number of enumerated value a Genesis value can take in the global section, \( G \) is the number in the generate statement, and \( E_P \) is the number of enumerated value a Genesis value can take in the program section.

Each program set may not contain the same the number of programs when using \texttt{enumerate}. For example,
In this example, the number of possible values \texttt{enumValue} enumerates through is unknown until \texttt{upperBound} is sampled. If this code is in the global section, \texttt{upperBound} is set once, and each enumerated value of \texttt{enumValue} generates 5 programs. However, if these enumerated values are put in the program section instead, 5 instance program sets are generated, with the number of programs in each set sampled independently.

When this happens, the total number of programs generated is:

\[
N = E_{P1} + E_{P2} + \ldots + E_{Pn} \quad \text{(where } n = E_G \ast G)\]

where \( N \) is the total number of programs generated, \( E_G \) is the number of enumerated value a Genesis value can take in the global section, \( G \) is the number in the generate statement, and \( E_{Pi} \) is the number of enumerated value a Genesis value can take in the program section during the \( i \)th iteration.

### 5.3 Using \texttt{genif}

The \texttt{genif} construct allows users to conditionally generate code. When used alongside Genesis' ability to sample values, \texttt{genif} constructs increase the possible diversity of instance programs in a set.

Figure 5.6 shows a Genesis program example demonstrating a use of \texttt{genif} in the context of a possible loop interchange of a pair of nested loops. In this example, inside the \texttt{loop} feature, \texttt{ifInterchange} is sampled on line 8 from a binary distribution of 0 and 1. Using \texttt{genif} constructs on lines 9 and 10, the value of \texttt{ifInterchange} controls the ordering of the loops. The \texttt{loop} feature is referenced in the template program shown in Figure 5.7 on line 3.
begin genesis

feature loopBody
  ...
end

feature loop
  value ifInterchange sample binaryDist
  genif ${ifInterchange}==0
    value inValue sample randomDist
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        ${loopBody} with ${inValue}
      }
    }
  end
  genif ${ifInterchange} == 1
    value inValue sample randomDist
    for (int j = 0; j < m; ++j) {
      for (int i = 0; i < n; ++i) {
        ${loopBody} with ${inValue}
      }
    }
end

generate 5 with randomDist = {1:10}, binaryDist = {0:1}
end genesis

Figure 5.6: Genesis Program using genif

void genifExample(int number){
  ${loop}
}

Figure 5.7: Template Program referencing loop
When the Genesis and template programs are read by the preprocessor, the \texttt{genif} statements are stored during the parsing phase. Genesis entities do not yet have a sampled value at this point, and thus, \texttt{genif} statements using Genesis names cannot yet be evaluated to be true or false at this time.

During program generation, for each instance program, \texttt{ifInterchange} samples a value when the \texttt{loop} feature is processed. That sampled value is used to evaluate the conditions of each \texttt{genif} statement, determining which code block is used in each instance program. An example of one instance program, where \texttt{ifInterchange} has sampled 0, is shown in Figure 5.8. The instance programs generated do not contain any if statements in the target language, as these \texttt{genif} statements are only used to control the flow of processing.

\section{5.4 Using \texttt{genloop}}

The \texttt{genloop} construct further increases the usefulness of Genesis. The construct allows users to repetitively generate code, removing unnecessary repetitive coding. These constructs are also useful when the bound of the loop is a sampled Genesis value. This value varies between programs and thus, cannot be hard-coded in a Genesis program. The construct also provides an iterator value that is available for use in the \texttt{genloop}. Figure 5.9 shows a Genesis program demonstrating the use of the \texttt{genloop} construct along with feature arguments.
In the feature `functionBody`, on line 13, `numAccesses` samples a value from 1 to 10. The `genloop` statement on line 14 uses this value as a bound. While the `genloop` processes, `loopIter` iterates through the integer values from 1 to `numAccesses`. The `loopIter` iterator acts as a Genesis value and is then passed into the `access` feature as an argument on line 15. This argument determines the offset for the array access on line 9.

The corresponding template program, a simple function containing a reference to `functionBody`, is shown in Figure 5.10.

An example of one instance program generated from the Genesis and template programs is shown in Figure 5.11. The value `numAccesses` samples 9, and thus, 9 feature references to `access` is generated, each one replaced with a new feature instance. Once again, the instance programs generated do not contain any loop statements in the output language, as this `genloop` statement is only used to control the flow of processing.
1   void genloopExample(int number){
2     temp5 = arr[1];
3     temp1 = arr[2];
4     temp3 = arr[3];
5     temp2 = arr[4];
6     temp5 = arr[5];
7     temp8 = arr[6];
8     temp6 = arr[7];
9     temp9 = arr[8];
10    temp6 = arr[9];
11  }

Figure 5.11: Example Instance Program generated from genloop Example

5.5 Using Processing Order

In previous sections, examples were used where the ordering of Genesis statements can affect the processing of the Genesis program. For example, in the add and remove example in Section 4.8, the code was processed sequentially, causing the use of remove affecting the later sampling of the Genesis variable dest2. In the enumerate example in Section 5.2, having the sample construct for the first value caused it to hold its value while the second value was enumerating. This section demonstrates an example where ordering affects distributions that use Genesis values.

For simplicity, the template program for this example is a simple program containing a single reference to feature limitExample:

void orderingExample(int number){
  ${limitExample}
}

Figure 5.12 shows the Genesis program that defines limitExample. In this feature definition, on lines 8-21, lowerLimit and upperLimit are defined as 2 Genesis values immediately set to 1 and 5, respectively, on lines 9-10. These two values are used to then define a distribution firstDist on line 11. The Genesis value oldSample1 is then sampled from this distribution on line 12. Thus, oldSample1 samples a value from 1 to 5 when processed.

Afterwards, on lines 13-14, the values of lowerLimit and upperLimit are changed.
### Figure 5.12: Genesis Program demonstrating effects of the Processing Order

Because these changes happen after `firstDist` is defined, its definition is not changed despite the Genesis values it depends on changing. Thus, when `oldSample2` is sampled from this distribution on line 15, it once again samples a value from 1 to 5, as defined previously.

On line 16, a second distribution `secondDist` is defined in the same way as `firstDist`, using the values of `lowerLimit` and `upperLimit` for its bounds. However, in contrast, `secondDist` uses the new values of `lowerLimit` and `upperLimit`, creating a distribution from 15 to 20. Thus, `newSample` sampling from this distribution takes on a value from 15 to 20. Lines 18-20 output the results of the samplings to the instance program.

An example instance program generated when this Genesis program is run is shown in Figure 5.13. The 3 lines of code generated by `limitExample` are written to the program, with the resultant Genesis values replaced by their sampled values.
**Figure 5.13: Example Instance Program from Processing Order Example**

```plaintext
void orderingExample(int number){
    print "This value is between 1 and 5: 5"
    print "This value is between 1 and 5: 1"
    print "This value is between 15 and 20: 16"
}
```

**Figure 5.14: Genesis Program using all sections of Genesis**

```plaintext
begin genesis

global
    value globalValue sample sampleDist
end

program
    value setValue sample sampleDist
    value enumerator enumerate enumeratorDist
    value holdValue sample sampleDist
end

feature varSet
    value featureValue sample sampleDist
    ## This set of code snippets will be
    ## outputted in each instance
    ## program with Genesis names replaced.
    #SAME ALWAYS: ${globalValue}
    #SAME THROUGH SET: ${setValue}
    #ENUMERATED: ${enumerator}
    #DIFFERENT PER PROGRAM: ${holdValue}
    #DIFFERENT IN PROGRAM: ${featureValue}
end

generate 2 with sampleDist = {1:100}, enumeratorDist = {1:2}
end genesis
```
5.6 Using the Sections of Genesis

As stated in Section 3.1, the location of a Genesis entity’s declaration affects the length of time an entity keeps its sampled value. Figure 5.14 shows a Genesis program example where values are declared in different sections.

In this example, `globalValue` is defined in the global section on line 4, some Genesis values are defined in the program section on lines 8-10, and `featureValue` is defined in a feature `varSet` on line 14. Each sampled entity is referenced inside `varSet` on lines 18-22, each replaced with its sampled value when processed. With the `generate 2` statement on line 25, along with the value `enumerator` in the program section on line 9 enumerating through 2 values, 4 instance programs are generated in 2 sets of 2 programs.

The template program for this example is a simple program containing a reference to feature `varSet`, repeated twice:

```java
void sectionExample(int number){
    ${varSet[2]}
}
```

Each processing of `varSet` results in a new sampling of `featureValue`, defined in the feature.

The flow of processing the Genesis program and template program is shown in Figure 5.15. First, as shown under “Process Global Section”, the global value `globalValue` is sampled once and held constant through all 4 instance programs. Next, under “Process Program Section”, `setValue` is sampled once per program set. While that value is held constant, the enumerator `enumerator`) generates two program sets. For each value of `enumerator), `holdValue` is sampled independently for each set. Next, while working under “Process Template Program”, `featureValue` is sampled once for each feature reference to `varSet`. Thus, `featureValue` can be different in each different feature reference within the same instance program.

An example of four generated instance programs in a set is shown in Figure 5.16.
These four instance programs demonstrate the difference between values declared in the global/program section and in individual features. The global and program sections are processed at predetermined times, while the features were only sampled when it was determined that the feature reference actually exists, each value being held constant for a different length of time.

5.7 Using \texttt{enumerate} with Stored Features

Using enumeration and stored Genesis entities, such as values or stored features, in different orders control the number of programs those entities are stored across. Using enumeration and stored features together can then be useful for applications that make use of optimizations, such as loop unrolling. In these cases, randomly sampled values used as characteristics in a program space can be held constant for multiple programs while enumerating through the possible optimization values. For example, a set of sampled
void sectionExample(int number) {
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 77
  # ENUMERATED: 1
  # DIFFERENT PER PROGRAM: 26
  # DIFFERENT IN PROGRAM: 26
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 77
  # ENUMERATED: 2
  # DIFFERENT PER PROGRAM: 44
  # DIFFERENT IN PROGRAM: 94
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 77
  # ENUMERATED: 2
  # DIFFERENT PER PROGRAM: 44
  # DIFFERENT IN PROGRAM: 17
}

(a) Program 1 in Program Set

void sectionExample(int number) {
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 36
  # ENUMERATED: 2
  # DIFFERENT PER PROGRAM: 53
  # DIFFERENT IN PROGRAM: 87
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 36
  # ENUMERATED: 2
  # DIFFERENT PER PROGRAM: 53
  # DIFFERENT IN PROGRAM: 36
}

(b) Program 2 in Program Set

void sectionExample(int number) {
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 36
  # ENUMERATED: 2
  # DIFFERENT PER PROGRAM: 53
  # DIFFERENT IN PROGRAM: 36
}

(c) Program 3 in Program Set

void sectionExample(int number) {
  ## This set of code snippets will be
  ## outputted in each instance
  ## program with Genesis names replaced
  # SAME ALWAYS: 37
  # SAME THROUGH SET: 36
  # ENUMERATED: 2
  # DIFFERENT PER PROGRAM: 53
  # DIFFERENT IN PROGRAM: 36
}

(d) Program 4 in Program Set

Figure 5.16: Example Instance Program Set demonstrating each section
characteristics can be held constant through all loop unrolling factors, creating a set of programs with the same characteristics spanning those factors.

Consider the following partial code example:

```plaintext
value stride1 sample {0:7}
value stride2 enumerate {0:7}
feature storedComp process computation
```

In this example, the value `stride1` is sampled, and held constant while `stride2` is enumerating. For each value of `stride2`, a new `computation` feature is processed and stored in `storedComp` for use later. The 8 different enumerated values will cause the preprocessor to generate 8 different `computations` stored in `storedComp`, one for each instance program in a set.

In contrast:

```plaintext
value stride1 sample {0:7}
feature storedComp process computation
value stride2 enumerate {0:7}
```

This code example results in `storedComp` being held constant while `stride2` is enumerating, since the processing of `storedComp` happens before the `enumerate` construct. Each program in a set has 1 of the 8 different enumerated values, but have the same stored `storedComp` feature. In this case, `stride2` cannot be used in `storedComp`, since `stride2` has not yet been given a value during `storedComp`'s processing.

Based on these two variations, ordering controls when stored features are processed and how long these features are stored, resulting in a different set of instance programs.

### 5.8 Using Multiple `enumerate` Statements

Figure 5.17 shows a simplified Genesis program example demonstrating how Genesis handles multiple uses of `enumerate` in the program section. The template program is a single reference to a feature `outputCode`:
begin genesis

global

program

value setValue sample sampleDist
value firstEnumerator enumerate enumeratorDist
value secondEnumerator enumerate enumeratorDist
value holdValue sample sampleDist
value endEnumerator enumerate enumeratorDist
end

feature outputCode

## This set of code snippets will be
## outputted in each instance
## program with Genesis names replaced
SAME THROUGH SET: ${setValue}
ENUMERATED: ${firstEnumerator}
ENUMERATED: ${secondEnumerator}
DIFFERENT PER 4 PROGRAMS: ${holdValue}
ENUMERATED: ${endEnumerator}
end

generate 1 with sampleDist = {1:100}, enumeratorDist = {1:4}
end genesis

Figure 5.17: Genesis Program using Multiple enumerates

```java
void multiEnumExample(int number){
    ${outputCode}
}
```

The `generate 1` statement on line 24 in the Genesis Program results in a single program set. There are three `enumerate` statements in the program section on lines 8, 9, and 11. Since each of these 3 Genesis values enumerates through 4 possible values, the preprocessor generates 4x4x4 or 64 instance programs in the set. These values, as well as two other Genesis values on lines 7 and 10, are output once in each instance program through the value references in the `outputCode` feature on lines 14-27.

The flow of the preprocessor is shown in Figure 5.18. First, the program section is processed sequentially. The value `setValue` on line 7 samples a value that is held constant for the entire program set. `firstEnumerator` enumerates on line 8, and first takes on a value of 1, holding it constant while the preprocessor continues processing every line after it. After, `holdValue` samples a value on line 10, and the other two enumerating Genesis values on lines 9 and 11 also hold their first values constant. When `endEnumerator` is
set to 1 on line 11, the program section has finished processing for the current instance program.

The preprocessor then begins searching the template program for feature references, and detects the `outputCode` reference. It processes this feature, makes all the value replacements, and then replaces the feature reference with the processed feature instance. An example of the resultant instance program is shown in Figure 5.19a.

Since the Genesis values in the program section still require enumerating, the preprocessor rolls back, and discovers that the Genesis value `endEnumerator` is still enumerating. Thus, it is next set to 2, while all other values before this line remain the same as before. The second instance program is then generated, shown in Figure 5.19b. Compared to Figure 5.19a, only the enumerated value of `endEnumerator` has changed.

When `endEnumerator` finishes enumerating through the 4 values in its assigned distribution and 4 instance programs are generated, the preprocessor detects that this Genesis value has finished enumerating, and thus it continues to roll back. It discovers that `secondEnumerator` on line 9 has not finished enumerating and sets it to 2.
### 5.9 Using Library Files

Genesis uses library files to implement some Genesis features that may be useful across many instance programs with the same output language. Ideally, the features included in a library file are the same for every Genesis program for a target language, removing the need for a user to repeatedly write these features. Thus, several library files containing implemented features for use in some languages are provided as part of Genesis. These
library files are added to a Genesis program using the geninclude construct.

Figure 5.20 shows an example library file, gen_c.glb, with features defined using regular Genesis constructs written for C programs. If this library file is included in a Genesis program, the two features in the file are available for use in the program like any other feature.

The feature keepLive, defined on lines 1-10 of Figure 5.20, generates C code summing up the variables in an indicated varlist and stores the result in some destination variable. This feature is intended to be used to keep these variables alive at the end of a block of code in the instance programs.

Two arguments are passed into the feature. The first argument, dest, is the name of the destination variable. The second argument, varlist is the name of the varlist with the variables to keep alive. Lines 3-8 use a genloop construct that sums through all the variables in the varlist. The reference inside of a reference on line 3 gives the size of the varlist. The first reference is the passed in argument varlist. This reference is replaced by the name of a varlist. The outer reference using that name and the size argument replaces the reference with the size of the varlist. This size is then used as a bound for the genloop. Lines 5-7 use a genif construct, which adds the plus symbol after each variable for every iteration of the genloop except the last, which then adds a semicolon on line 9.
begin genesis

geninclude gen_c.glb

program
    varlist twentyList[20] name(temp)
    variable sampledVar from twentyList
end

feature outputCode
    print "Here is some code using ${sampledVar}"
    ${keepLive(*result, twentyList)}
end

generate 1000
end genesis

Figure 5.21: Example of the Use of a Library File

The `singleline` modifier on line 1 compresses the processed feature to be returned as a single line of C code.

Another feature, `varlistDeclare`, defined on lines 12-16, initializes variables in the code. The type and name of the varlist and are passed in as arguments to the feature. This feature uses a genloop similar to the one in `keepLive` to loop through each variable in the varlist argument and generate C initialization code for each one. Each variable is initialized in C with the type passed in as an argument to the feature.

An example of the use of one of the features in this library is shown in Figure 5.21. The `geninclude` construct on line 3, when processed, substitutes the contents of the file to replace the line, and thus, the two features in `gen_c.glb` are available for use. On line 12, `keepLive` is used, which was defined in the library file, and thus was allowed to be used as a feature reference. `*result` is passed in as the name of the destination variable, assuming it is a valid string used in the template program. `twentyList` is the name of the varlist, originally declared on line 6, whose variables are to keep alive. When the feature `outputCode` is processed, this reference to `keepLive` is replaced by C code that keeps the variables in `twentyList` alive by storing the sum result into `*result`. 
Chapter 6

Case Studies

In this chapter, we present three case studies to demonstrate the utility of Genesis. The case studies come from different application domains where having a large number of synthetically generated instance programs can be beneficial. Section 6.1 gives an example from the domain of the auto-tuning of image filtering applications. Section 6.2 gives an example of generating programs with different static program characteristics. Section 6.3 gives an example in the domain of image recognition.

6.1 Case Study 1: Image Filtering

For the first case study, we consider image filtering applications. These applications are generally highly parallelizable, making them perfectly suited on GPUs [28]. An application can be optimized to further decrease its runtime on GPUs. However, these optimizations can also slow down the application. Thus, the decision to use a specific optimization is non-trivial [28, 36, 56]. Testing all combinations of optimizations in an exhaustive search for the best configuration is an infeasible, time-consuming process.

Machine learning can be used to automatically make this prediction [28]. As described in Section 2.2.1, this performance auto-tuning requires a large amount of applications for use in training a ML model. These applications can be generated synthetically.
Many image filtering applications operate on two dimensional images. The computationally intensive component of these applications have typically have two nested loops that sweep over the two-dimensional images. Each output element, or pixel, is computed as a function of a subset of the pixels in an input image. Specific image filtering applications differ in the subset of input pixels and the function used to compute the output pixels.

For this case study, we model the possible image filtering applications in this space as a set of applications which all contain two perfectly nested loops. The body of each loop nest contains one or more read epochs followed by a write epoch, where an epoch is a block of code containing a sequence of computations followed by a memory access. A computation consists of three source variables used in a product-sum and the result placed in a destination variable. A memory access consists of an access to one of two one-dimensional arrays. The type of access, read or write, determines if it reads from the input array or writes to the output array. It is desired to generate applications where the number of read epochs, the number of computations per epoch, and the pattern of memory accesses all vary.

Genesis provides the means of generating a large amount of applications for this purpose. Figure 6.1 shows the Genesis program that describes these applications. It defines five features, each defining one of the terms above. The first feature, on lines 19-22, defines a computation, which samples four different variables from the varlist temp. The code snippet in this feature definition computes a product-sum using three of the sampled variables, and assigns it to the variable sampled by dest.

The read_access feature on lines 24-29 defines a memory read that samples a destination variable and three values. The three values and the loop iterators (it0 and it1) determine the input array element to read, storing the accessed value in the destination variable. Similarly, the write_access feature on lines 31-35 defines a memory write, where a variable is stored in an output memory location determined by the loop iterators,
begin genesis

geninclude gen_c.glb

global
distribution epochdist = {1:10}
distribution numvardist = {8;16;32}
distribution compDist = {1:20}
distribution coefdist = {0:7}
distribution offsdist = {0:15}
end

program
value numEpochs sample epochdist
value numVars sample numvardist
varlist temp[$(numVars)]
end

feature computation
variable dest,src1,src2,src3 from temp
$dest = $src1 * $src2 + $src3;
end

feature read_access
variable dest from temp
value coef1, coef2 sample coefdist
value offs_r sample offsdist
$dest = arr_in[$(coef1)*it0 + $(coef2)*it1 +$offs_r]];
end

feature write_access
variable src from temp
value offs_w sample offsdist
arr_out[inner_tc*it0 + it1 +$offs_w] = $src;
end

feature epoch (epoch_type)
value numComps sample compDist
genloop i:1:$numComps
$computation
end
genif $(epoch_type) eq "read"
$read_access
genelse
$write_access
end
end

feature loopbody
genloop i:1:$numEpochs
$epoch("read")
end
$epoch("write")
end

generate 1000
end genesis

Figure 6.1: Image Filtering Genesis Program (IFGP)
void filter(unsigned int outer_tc, unsigned int inner_tc, global float *arr_in, 
global float *arr_out, global int *result){
  ${varlistDeclare(int, temp)}
  for (int it0 = get_local_id(0); it0 < outer_tc; it0 += get_local_size(0)) {
    for (int it1 = get_local_id(1); it1 < inner_tc; it1 += get_local_size(1)) {
      ${loopbody}
    }
  }
  ${keepLive(*result, temp)}
}

Figure 6.2: Template Program for Image Filtering

the inner trip count, and a sampled offset value.

With these three building blocks, an epoch feature is defined. This feature, defined on lines 37-47, defines both reads and write epochs, distinguished by the argument passed into the feature. It consists of multiple computations followed by a read or write access. The value numComps is sampled and, using a genloop, the computation feature is referenced numComps times. After the set of computations is either a read_access or a write_access depending on the value of the epoch_type argument. Thus, this definition describes all the possible epoch feature instances in the program space.

The fifth and final feature loopbody, defined on lines 49-54, is a feature containing multiple references to the epoch feature. This feature consists of numEpochs read epochs, followed by a single write epoch. Each read epoch is created by passing in a “read” argument to the epoch feature, while each write epoch is created by passing in a “write” argument. The value numEpochs is sampled in the program section. Thus, this value is sampled once for each of the generated instance programs, likely resulting in a different number of epochs in each program. Thus, this definition describes all the possible loopbody feature instances in the program space.

The template program, shown in Figure 6.2, is a skeletal OpenCL application that contains the loops that sweep over the image and a loopbody reference. The template program also contains two references to features defined in the library gen_c.glb:
varlistDeclare, which initializes variables in a varlist, and keepLive, which touches every element in a varlist to keep it live and writes to the supplied location.

Figure 6.3 shows an example of one of the instance programs that can be generated from processing the Genesis and template programs. In this instance program, the feature reference of loopbody is replaced by a processed feature instance. This feature instance contains processed feature instances of all the features referenced in its definition as well. The references to varlistDeclare and keepLive were replaced by processed feature instances, generated from the feature definitions in the existing library. The end result, with the generate 1000 line, is the creation of 1000 instance programs, each consisting of multiple read epochs and a write epoch. Each instance program contains a variable number of epochs, number of computations in each epoch, and pattern of memory accesses.

6.2 Case Study 2: Static Program Characteristics

The second case study is inspired by work done by cTuning on the auto-tuning of programs. Its MilepostGCC compiler [22] extracts characteristics from a program [19], and uses the information and a ML model to tune the program for performance. Many characteristics come from low-level properties of the intermediate representation of a program such as the number of basic blocks, the number of instructions per basic block, the number of back edges, and the number of basic blocks with two successors [19]. Thus, the goal of this case study is to use Genesis to generate a large number of programs with varying values of these characteristics as inputs to this tuning problem.

For simplicity, we focus only on varying the type and number of instructions per basic block and the number of basic blocks. C is used for higher readability, while still allowing a one-to-one correspondence to these low-level properties. Each program contains a sequence of basic blocks. Each block has a series of C low-level instructions, consisting
### Figure 6.3: Instance Program for Image Filtering

```c
// Creating with ./spec - demo1000 .c
// File created: gen16_1 .c

void filter(unsigned int outer_tc, unsigned int inner_tc, __global float *arr_in,
            __global float *arr_out, int *result){
    int temp1 = 0;
    int temp2 = 0;
    int temp3 = 0;
    int temp4 = 0;
    int temp5 = 0;
    int temp6 = 0;
    int temp7 = 0;
    int temp8 = 0;

    for (int it0 = get_local_id(0); it0 < outer_tc; it0 += get_local_size(0)) {
        for (int it1 = get_local_id(1); it1 < inner_tc; it1 += get_local_size(1)) {
            temp5 = temp2 * temp2 + temp6;
            temp3 = arr_in[7*it0 + 5*it1 +5];
            temp7 = temp3 * temp5 + temp7;
            temp8 = temp1 * temp4 + temp6;
            temp1 = temp3 * temp6 + temp4;
            temp1 = temp3 * temp6 + temp7;
            temp3 = temp8 * temp6 + temp8;
            temp4 = temp3 * temp6 + temp4;
            temp7 = temp2 * temp2 + temp5;
            temp7 = temp7 * temp5 + temp6;
            temp4 = temp6 * temp1 + temp1;
            temp7 = temp7 * temp5 + temp6;
            temp8 = arr_in[5*it0 + 6*it1 +2];
            temp2 = temp6 * temp5 + temp3;
            temp3 = temp8 * temp6 + temp3;
            temp6 = temp3 * temp8 + temp4;
            temp2 = temp1 * temp1 + temp4;
            temp5 = temp7 * temp8 + temp1;
            temp8 = temp3 * temp3 + temp6;
            temp3 = temp2 * temp2 + temp8;
            arr_out[inner_tc*it0 + it1 +14] = temp5;
        }
    }
    *result = temp1 + temp2 + temp3 + temp4 + temp5 + temp6 + temp7 + temp8 ;
}
```
Figure 6.4: Template Program for Static Program Characteristics

of sum, copy or load-from-memory instructions. Each block ends with a goto statement to the next basic block. Thus, this case study is a first step towards generating a synthetic training set for MilepostGCC or similar projects.

The template program for these programs is shown in Figure 6.4. It consists of a simple function definition, with a reference to a codebody feature.

Figure 6.5 shows the Genesis program that can be used to generate 1000 instance programs from the template program. The instructions that can be sampled are defined in multiple features on lines 14-32, and consist of a sum instruction, a copy instruction, and a load instruction. The can_be_defined varlist keeps a list of temp variables that were used in the instance program and can be sampled by the Genesis variable dest. The add and remove constructs in the instruction features manipulate can_be_defined to ensure that no dead code is produced. Variables are removed from the varlist once used in the instance program as a source. Once removed, the sampled value is no longer allowed to be defined. The sampled value is re-added to this varlist after being used in the instance program and are allowed to be defined once again.

The instruction sampling is performed in the singleInsn feature on lines 34-43, where a random instruction type is referenced using a value sampled from insn_type_dist performed on line 35.

The top level feature, codeBody, is defined on lines 45-58. The numBlocks value is sampled and determines the number of basic blocks that are in the instance program. The outer genloop construct creates a basic block with each iteration. The numInsns value is sampled in each iteration of the outer genloop, and determines how often singleInsn is repeated by the inner genloop. The goto instruction on line 55 explicitly splits up the
begin genesis
  global
distribution insn_type_dist = {"sum","cp","ld"}
distribution insns_dist = {1:20}
distribution bb_dist = {2:5}
distribution offs_dist = {0:7}
end
program
  varlist temp[5]
  varlist can_be_defined from temp
end

feature suminsn
  variable src1, src2 from temp
  add src1, src2 to can_be_defined
  variable dest from can_be_defined
  remove dest from can_be_defined
  \$\text{dest} = \$\text{src1} + \$\text{src2};$
end

feature cpinsn
  variable src from temp
  add src to can_be_defined
  variable dest from can_be_defined
  remove dest from can_be_defined
  \$\text{dest} = \$\text{src};$
end

feature ldinsn
  value offs sample offs_dist
  variable dest from can_be_defined
  remove dest from can_be_defined
  \$\text{dest} = \text{arr}[\$\text{offs}];$
end

feature singleinsn
  value insntype sample insn_type_dist
  genif \$\text{insntype} eq "sum"
    \$\text{suminsn}$
  genelsif \$\text{insntype} eq "cp"
    \$\text{cpinsn}$
  genelsif \$\text{insntype} eq "ld"
    \$\text{ldinsn}$
end

feature codebody
  value numBlocks sample bb_dist
  genloop loopVar:1:${numBlocks}
    T$\text{loopVar}$:
      value numInsns sample insns_dist
      genloop insn:1:${numInsns}
        \$\text{singleinsn}$
      end
      genif \$\text{loopVar}$!=${numBlocks}
        value dest = \$\text{loopVar}$+1
        \$\text{gotoinsn}($$\text{dest}$$)$
      end
    end
  end

feature gotoinsn(d)
  goto T$\text{dest}$;
end

generate 1000
end genesis

Figure 6.5: Static Program Characteristics Genesis Program (SPCGP)
void basic_block_code(float *arr) {
    int temp1 = arr[0];
    int temp2 = temp1;
    int temp3 = temp2;
    int temp4 = temp3;
    int temp5 = temp4;
    T1:
    temp3 = arr[3];
    temp5 = temp1 + temp4;
    temp1 = temp1;
    temp5 = temp5 + temp1;
    temp1 = temp4 + temp5;
    temp4 = temp2 + temp2;
    temp5 = arr[7];
    temp2 = temp2 + temp1;
    goto T2;
    T2:
    temp1 = arr[3];
    temp1 = temp1 + temp3;
    temp3 = temp3 + temp4;
    goto T3;
    T3:
    temp4 = arr[7];
    temp1 = temp1;
    temp5 = temp5 + temp5;
    temp4 = temp4;
    temp1 = temp1;
    }

Figure 6.6: Genesis Output for Static Program Characteristics

basic blocks in each linear program. This top level feature of codeBody describes all the possible codeBody feature instances in our space, and is the only feature referenced in the template program.

Figure 6.6 shows a possible instance program generated by the preprocessor with these programs. The entire reference to codeBody is replaced by the entire processed feature, with each block containing multiple randomly sampled instructions. Each block begins with a label and ends with a goto statement, splitting up each block explicitly.

A limitation with the Genesis program shown in Figure 6.5 is the possibility that the can_be_defined varlist can run out of variables. If this happens, and a load instruction is sampled, sampling from can_be_defined on line 29 fails. In this case, the instance program fails to generate and is skipped over. Thus, a possibility exists that not all 1000 will successfully generate. The other two instructions do not have this problem, as variables are added to the can_be_defined varlist in these features before the varlist is sampled from.
begin genesis
    global
distribution insn_type_dist = {"sum","cp","ld"}
distribution insn_type_dist2 = {"sum","cp"}
distribution insns_dist = {1:20}
distribution bb_dist = {2:5}
distribution offs_dist = {0:7}
end

program
    varlist temp[5]
    varlist can_be_defined from temp
end

feature suminsn
    variable src1, src2 from temp
    add src1, src2 to can_be_defined
    variable dest from can_be_defined
    remove dest from can_be_defined
    ${dest} = ${src1} + ${src2};
end

feature cpinsn
    variable src from temp
    add src to can_be_defined
    variable dest from can_be_defined
    remove dest from can_be_defined
    ${dest} = ${src};
end

feature ldinsn
    value offs sample offs_dist
    variable dest from can_be_defined
    remove dest from can_be_defined
    ${dest} = arr[{$offs}];
end

Figure 6.7: Modified Static Program Characteristics Genesis Program (SPCGP2) (Part 1 of 2). The other half of the program is in Figure 6.8.
Figure 6.8: Modified Static Program Characteristics Genesis Program (SPCGP2) (Part 2 of 2). The first half of the program is in Figure 6.7.
One way of solving this limitation is shown in Figure 6.7 and Figure 6.8. A new distribution is defined on line 4. The singleInsn feature on line 37 is extended to prevent the case of an empty can_be_defined varlist. In this case, described on lines 38-44, load cannot be sampled and the empty varlist cannot be sampled from.

The downside of this modified Genesis program is that, since loads will appear less often, the likelihood of each instruction is no longer one in three. However, it prevents any instance programs from failing to generate, and all 1000 programs will generate successfully.

6.3 Case Study 3: Image Layering

The last case study is motivated by image recognition software, i.e., software that detect faces or shapes in an image [41]. A large set of images with varying amounts of faces and shapes of different sizes can be used to test how well software algorithms work at detecting faces and shapes. Also, many image recognition software algorithms, such as the ones used in OpenCV [41], use machine learning techniques [41], and thus requires a large amount of input images that the technique will learn from. It can be difficult to find a diverse set of images where the faces and shapes vary in their size and location. Thus, a large amount synthetically generated images can instead be used to compensate and help improve the quality of image recognition software.

Many image creation programs and tools, such as PhotoShop [30] and Paint Shop Pro [47] use superimposing techniques and layers to layer images on top of one another into a single image. Genesis can use this idea to support image generation. The benefit of Genesis is that the random sampling of the ordering and locations of image loading and placement during the layering process can be easily generated.

For simplicity, instead of generating the images directly, we assume that an image generation program uses a set of language statements shown below. These statements
control the loading and placement of images. Assume that this language consists of the following simple language statements:

- load (filename) to outputFile
- place (filename) at height (height) and width (width) with size (size)x
- store outputFile to (filename)

These constructs allow a user to create a layered photo of faces by generating commands that load an image as a background, overlay images on top of one another, and store the result to an output image. By using these language constructs, Genesis can generate a large number of images where each image varies in the location and size of the varying faces in question. These images scale up accordingly, and based on their locations, can end up partially occluding one another as faces are layered on top of one another.

Figure 6.9 shows the Genesis program used for this purpose. We assume that each background image is a 1024x1024 pixel image, but make no assumptions on the faces used to overlay.

The template program is a single line with a reference to the top-level feature `createImage`, indicating that the entire code should vary:

`${createImage}`

Since the template program contains no other code, each instance program will likely be different from one another, with no explicit identical lines.

The distributions are laid out in the global section on lines 3-9. These distributions control the ranges of the backgrounds used, the number of faces to overlay, the filename of the face to overlay, the locations the faces are placed on the image, and the size of the sampled face.

The feature definition of `createImage` on lines 38-43 contains four lines: a reference to the `loadImage` feature, a value `numberFaces` determining the number of faces to load, a reference to the `overlayFace` feature (using the sampled value `numberFaces` to indicate
CHAPTER 6. CASE STUDIES

begin genesis

global
distribution backgroundDist = {1:3}
distribution numFacesDist = {1:10}
distribution facesDist = {1:1000}
distribution locationDist = {0:1023}
distribution sizeDist = {1:10}
end

program
end

feature loadImage
value background sample backgroundDist
genif ${ background } == 1
genmath background = "grass.jpg"
genelsif ${ background } == 2
genmath background = "field.jpg"
genelsif ${ background } == 3
genmath background = "house.jpg"
end
load "${ background }" to outputFile
end

feature overlayFace
value heightValue sample locationDist
value widthValue sample locationDist
value sizeValue sample sizeDist
value face sample facesDist
place facefile$ { face }.jpg at height ${ heightValue } and width ${ widthValue } with size ${ sizeValue }x
end

feature storeImage
store outputFile to "output.jpg"
end

feature createImage
${ loadImage }
value numberFaces sample numFacesDist
${ overlayFace[ ${ numberFaces } ] }
${ storeImage }
end

generate 1000
end genesis

Figure 6.9: Image Layering Genesis Program (ILGP)
how often faces are overlaid), and a reference to the \texttt{storeImage} feature. The three features referenced are for loading an image, placing an face onto an image, and storing an image, respectively.

Loading an image as a background (feature \texttt{loadImage} on lines 14-24) is done by first sampling a value from \texttt{backgroundDist}. Depending on the sampled value, the filename from which the background is loaded varies.

The feature \texttt{overlayFace} on lines 26-32 is referenced multiple times in \texttt{storeImage}. This feature samples two locations, a \texttt{height} value and a \texttt{width} value. It also samples a \texttt{size} multiplier and a number to indicate which face to load. These values are then placed into the command snippet, and returned and replaced in \texttt{storeImage}. This feature is referenced multiple times to load and place multiple layered faces.

Code storing the image to file, generated by feature \texttt{storeImage} on lines 34-36, is performed at the end of the generated commands. The feature is defined as a single resultant code snippet with no references, and thus, is the same across all instance programs. The definition requires no sampling, showing that features do not need varying parts if the user does not wish to have any. If the user wishes for each instance program to have a different output filename, \texttt{storeImage} is modified to accomplish this by keeping a global counter value and using \texttt{genmath} to increment this counter after every reference. Using a value defined in the global section \texttt{counter}, a modified feature \texttt{storeImage} can look like the following:
load "field.jpg" to outputFile
place facefile890.jpg at height 645 and width 870 with size 1x
place facefile955.jpg at height 7 and width 111 with size 9x
place facefile153.jpg at height 0 and width 766 with size 6x
place facefile697.jpg at height 506 and width 597 with size 1x
place facefile390.jpg at height 579 and width 591 with size 9x
place facefile540.jpg at height 306 and width 399 with size 7x
place facefile238.jpg at height 803 and width 785 with size 6x
place facefile773.jpg at height 575 and width 402 with size 9x
place facefile245.jpg at height 268 and width 120 with size 5x
store outputFile to "output.jpg"

Figure 6.10: Genesis Output for Image Layering

feature storeImage
  genmath counter = ${counter}+1
  load outputFile to "output${counter}.jpg"
end

When the preprocessor reads the Genesis program and template program, it generates 1000 image layering instance programs as indicated by the generate statement. A sample instance program is shown in Figure 6.10. This program contains a single loading of the image, a series of placing images at randomly sampled locations, and a final store to an output image. This resultant instance program is fed into a later program stage of some image creation software to read these commands and perform the resultant actions.

Changes to the instance programs generated can be implemented with small changes to the Genesis program. For this modified set of instance programs, the maximum number of faces is increased to 20, the instance programs store each output image into a different file, and the commands for multiple output images are placed in each file.

Figure 6.11 shows a modified version that changes the instance programs with minimal changes to the Genesis program. The maximum number of faces on an image is increased to 20 by modifying the distribution numFacesDist on line 5. The feature storeImage now uses a counter for the output filename as described previously.

Also, with a corresponding change to the template program, 100 instance programs are generated, with all the commands in that single program for 10 output images, as
begin genesis

global distribution backgroundDist = {1:3}
distribution numFacesDist = {1:20}
distribution facesDist = {1:1000}
distribution locationDist = {0:1023}
distribution sizeDist = {1:10}
value counter = 0
end

program end

feature loadImage
value background sample backgroundDist
genif background == 1
genmath background = "grass.jpg"
genelsif background == 2
genmath background = "field.jpg"
genelsif background == 3
genmath background = "house.jpg"
load "${background}" to outputFile end

feature overlayFace
value heightValue sample locationDist
value widthValue sample locationDist
value sizeValue sample sizeDist
value face sample facesDist
place facefile${face}.jpg at height ${heightValue} and width ${widthValue} with size ${sizeValue}x
end

feature storeImage
genmath counter = ${counter}+1
load outputFile to "output${counter}.jpg"
end

feature createImage
${loadImage}
value numberFaces sample numFacesDist
${overlayFace[${numberFaces}]}
${storeImage}
end

generate 100
end genesis

${createImage[10]}

Figure 6.11: Modified Image Layering Genesis Program (ILGP2)
opposed to 1000 instance programs. The template program for this modified Genesis program is as follows:

```latex
\texttt{${\text{createImage}[10]}$}
```

This modified template program now references the feature `createImage` ten times, placing all the commands into a single instance program. Thus, for the 100 instance programs generated, each one will store commands to create 10 image-layered photos.
Chapter 7

Implementation

In this chapter, we discuss our implementation of the preprocessor. Section 7.1 discusses our implementation in Perl. Section 7.2 discusses logging in our implementation.

7.1 Perl Implementation Overview

Genesis was implemented as a standalone preprocessor in Perl, and thus, Genesis is not limited to a specific target language. Using a scripting language such as Perl, as opposed to a proper lexer and parser reduced development time while keeping the implementation flexible as the language evolved.

The preprocessor works in three phases. During the first phase of file parsing, the preprocessor reads a Genesis program and builds an internal representation of the constructs present. Each line is stored in a separate array based on its Genesis construct type, such as value or variable, and given a distinct ID. Each feature is stored in memory, with each Genesis line in that feature represented by the construct type and ID. The template program is also read and stored during this phase.

In the second phase of instance generation, the information stored is used to generate the desired number of instance programs. First, the global section is processed. Then, for each of the generated instance programs, the program section is processed and a copy
of the template program is created. The code in each copy is scanned for any feature references using regular expressions. For each feature reference, the feature is processed and using similar regular expressions, the resulting code snippet replaces the reference. Random sampling of Genesis entities is done using the `rand()` function provided by Perl. Once all feature references in the template program are detected and replaced, the instance program is written to a file in the third and final phase of file output. The last two phases are done iteratively to generate the set the instance programs.

The preprocessor can produce a comment block at the beginning of each generated file that includes the sampled values for each Genesis entity used to generate that instance program. Since Genesis is language agnostic, the user must provide a comment character when running Genesis to produce this comment block in each file. The preprocessor can also display statistical information, such as how often each value in a distribution is sampled for a Genesis entity across all instance programs. Using these values, the preprocessor can also output an analysis of the sampled values using Pearson’s chi-squared test [43], which helps the user of Genesis determine the amount of deviation an actual set of sampled values has from its declared distribution. This output and statistical analysis is used for our evaluation in Section 8.2.

In some cases, instance programs can fail to generate. For example, an instance program can fail to generate if a user attempts to sample from an empty varlist. When this happens, the program is not generated and that program instance number is skipped. The preprocessor then continues onto the next instance program. Our Perl preprocessor implementation reports the number of programs generated, the number of program sets, and which programs failed to generate.
7.2 Logging

Our implementation provides logging information to the terminal. In the default mode, it logs the type of each parsed line, such as value, variable or feature definition, during the parsing phase. At the end of this first phase, a quick summary of the parsed information is printed. During the generation phase, it reports the name of the instance program it is currently generating, and any warnings or errors that occur. At the end of this second phase, a summary of the instance programs is printed, and the user is informed of any failed instance programs and any warning messages.

Our implementation also provides a verbose mode and a simple mode that differ in the amount of logging information displayed compared to the default mode. The verbose mode prints more detailed information to the screen than in the default mode. During the parsing, more information is displayed to the user for each parsed line, including the name, ID, and any other relevant information from the line. During generation, all Genesis value or variable sampling information is displayed, as well as any feature reference found and their resultant code snippets. In the simple mode, the preprocessor lowers the outputting to just the final summary at the end of each phase.

Our implementation also provides a system to inform the user when something unexpected occurs, including a set of error and warning messages. An error message is displayed when a mistake occurs during processing, causing it to end immediately. When this happens, no instance programs are generated. A warning message is displayed for less fatal mistakes, and which processing can still continue. Examples include declaring a value at the end of a feature (and thus it is not used), or a failed instance program. If a warning occurs, the user is notified and the current instance program may be discarded. A list of all possible Genesis error and warning messages is included in Appendix A.3.
Chapter 8

Evaluation

In this chapter, we give an evaluation of Genesis in three ways. Section 8.1 evaluates the correctness of our Genesis implementation. Section 8.2 examines the statistical quality of data sampling of Genesis values and the amount of deviance the sampled data have from their declared distributions using the image filtering case study of Section 6.2. Section 8.3 evaluates the performance of our Perl preprocessor using all three case studies of Chapter 6.

We collect the runtime and sampling data of running Genesis programs and template programs through the preprocessor on an Intel Core i7-4930K CPU running at 3.40GHz, and 32GB of memory, running Perl 5.18.2.

8.1 Correctness

The first aspect of evaluation is to test for the correctness of our implementation. To do so, a set of test Genesis programs with known outputs were created. These tests are used for regression testing of the implementation.

For example, to test the correct functionality of the genif statements, multiple test Genesis programs were created with these statements used in multiple ways, and known outputs for each possible feature code-path. A condensed version of the Genesis program
and template program testing genif statements is shown in Figure 8.1. Each test is put into its own feature, where taking each code-path results in a different output. The template program is on lines 57-63, consisting of references to each test. The corresponding correct output after processing the Genesis and template programs is shown in Figure 8.2. The preprocessor should read the Genesis program, and during the generation phase, the functionality will display differently depending on the code-paths it enters for each test. The output displayed in Figure 8.2 indicates that this test has entered the correct code-paths.

The test Genesis programs were also created to purposely reach each possible error and warning test-path to determine if the proper messages are displayed to the user. A list of all possible Genesis errors and warnings is included in Appendix A.3.

### 8.2 Statistical Sampling

As stated in Section 7.1, the preprocessor keeps track of sampling counts for each Genesis value, keeping track of how often each value in a distribution is sampled. Figure 8.3 shows the distribution of sampling numEpochs 1000 times during one run of the image filtering case study. The declared distribution for this Genesis value was a uniform distribution from 1 to 10. Because the distribution is uniform, the expected counts are 100 for each value. Thus, our sampling deviates from the expected amounts. This deviation will differ during different runs of the same Genesis program.

We evaluate the amount of deviance the sampling distribution has compared to the declared distribution using the Pearson’s chi-squared goodness of fit test [43]. A chi-squared ($\chi^2$) test is an indicator of how well a sampled distribution differs from the declared distributions. In this test, a $\chi^2$ value is calculated from the samples, where a higher resultant $\chi^2$ value indicates a greater deviation from the declared distributions, and a lower value gives greater confidence that the sampling came from the desired distribution.
begin genesis
feature test0
  TEST0 - Goes into IF. should output AAA
  value zero sample dist0
  genif ${zero}==0
    AAA
  genelse
    BBB
  end
end
End of Test

double
feature test1
  TEST1 - Goes into ELSE. should output BBB
  value one sample dist1
  genif ${one}==0
    AAA
  genelse
    BBB
  end
End of Test

double
feature test2
  TEST2 - Goes into ELSE with a genelsif. should output CCC
  value two sample dist2
  genif ${two}==0
    AAA
  genelsif ${two}==1
    BBB
  genelse
    CCC
  end
End of Test

double
feature test3
  TEST3 - Goes into 1st ELSEIF and 2nd GENIF. should output BBB DDD
  value three sample dist1
  genif ${three}==0
    AAA
  genelsif ${three}==1
    BBB
  genelse
    CCC
  end
genif ${three}==1
    DDD
  end
End of Test
end

generate 1 with dist0 = {0}, dist1 = {1}, dist2 = {2}
end genesis

${test0}
${test1}
${test2}
${test3}

Figure 8.1: Genesis and Template Programs for Genif Correctness Testing
CHAPTER 8. EVALUATION

Creating with ./SpecFiles/Testcases/spec-genif.c
File created: gen1_1.c

TEST0 - Goes into IF. should output AAA
AAA
End of Test

TEST1 - Goes into ELSE. should output BBB
BBB
End of Test

TEST2 - Goes into ELSE with a genelsif. should output CCC
CCC
End of Test

TEST3 - Goes into 1st ELSEIF and 2nd GENIF. should output BBB DDD
BBB
DDD
End of Test

Figure 8.2: Genesis Output for Genif Correctness Testing

Figure 8.3: Distribution of Sampling numEpochs
A calculated $\chi^2$ value can be converted to a P-value, the probability of observing a sample statistic as extreme as that $\chi^2$ statistic for that many degrees of freedom. The degrees of freedom is calculated as the number of possible outcomes in a distribution subtracted by one \([13]\). Because the P-value is a probability, the possible P-values uniformly range between 0 and 1. While other P-values can be used, 0.05 is an accepted threshold for significant deviance \([3]\). The deviance of a sampling with a P-value greater than 0.05 is considered reasonable, while a sampling with a P-value lower than 0.05 is expected to have some bias, as this sampling is in the 5% that deviate the most from a declared distribution. A $\chi^2$ value can be compared to a critical value, defined as the $\chi^2$ value that has a P-value of 0.05 \([13]\) taken from a lookup table using the degrees of freedom. Thus, samplings that have a $\chi^2$ higher than the critical value have a P-value lower than 0.05.

We conduct Pearson’s chi-squared test for the image filtering case study. Table 8.1 lists the case study’s Genesis values, their corresponding distributions and the degree of freedom for each value. Genesis is run to generate a 1000 image filtering applications and the sampled values are collected. $\chi^2$ values are calculated from each sampling distribution to determine if the deviation is reasonable. For example, the sampling distribution from Figure 8.3 results in a $\chi^2$ value of 7.26, and with 9 degrees of freedom, the calculated P-value is 0.61. This means the probability that a chi-square statistic having 9 degrees of

### Table 8.1: Test Results for One Run of the Image Filtering Genesis Program

<table>
<thead>
<tr>
<th>Variable</th>
<th>Distribution</th>
<th>Deg. of Freedom</th>
<th>$\chi^2$</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numEpochs</td>
<td>Uniform 1-10</td>
<td>9</td>
<td>7.26</td>
<td>0.61</td>
</tr>
<tr>
<td>numComps</td>
<td>Uniform 1-20</td>
<td>19</td>
<td>21.88</td>
<td>0.29</td>
</tr>
<tr>
<td>numVars</td>
<td>256, 512, 1024</td>
<td>2</td>
<td>3.30</td>
<td>0.19</td>
</tr>
<tr>
<td>coef1</td>
<td>Uniform 0-7</td>
<td>7</td>
<td>3.57</td>
<td>0.83</td>
</tr>
<tr>
<td>coef2</td>
<td>Uniform 0-7</td>
<td>7</td>
<td>5.83</td>
<td>0.56</td>
</tr>
<tr>
<td>offs_r</td>
<td>Uniform 0-15</td>
<td>15</td>
<td>19.76</td>
<td>0.18</td>
</tr>
<tr>
<td>offs_w</td>
<td>Uniform 0-15</td>
<td>15</td>
<td>10.88</td>
<td>0.76</td>
</tr>
</tbody>
</table>

without bias.
freedom being more extreme than 7.26 is 0.61. Since this probability is greater than 0.05, we conclude that a bias in the sampling is unlikely, and the sampling, while having some deviance, is reasonable. The same applies to the rest of the case study Genesis values for this run, shown in Table 8.1.

Table 8.2 shows the results of collecting the $\chi^2$ values over 10 runs, gathering 70 test results. The maximum and minimum $\chi^2$ values calculated, the critical value of each distribution, and any calculated $\chi^2$ values over the critical value are shown. This table shows that most of the $\chi^2$ values are below the critical value.

Figure 8.4 shows the distribution of the P-values calculated from the 70 $\chi^2$ tests. This graph shows a wide range of P-values, and visually, it appears that the distribution of P-value is reasonably varied. This means each set of programs Genesis generates varies in its degree of deviance, as expected when it comes to random sampling.

Through 10 runs, 66 out of 70 $\chi^2$ values, or 94.3%, ended up below their critical value, meaning a reasonable sampling. This approximately follows the definition of the chi-squared test, leading us to conclude that the statistical quality of sampling from distributions is as expected.
8.3 Preprocessor Performance

We also evaluate the performance of our Perl preprocessor. For each case study, we generate instance programs in powers of 10 from 10 to 100,000 using the five Genesis and template programs of the three case studies in Chapter 6, and measured the runtime of each run of the preprocessor. Each experiment was run 10 times and results are averaged.

Figure 8.5 shows the runtime as a function of the number of generated instance programs for each case study, where IPGP refers to the image filtering Genesis program in Figure 6.1, SPCGP and SPCGP2 refer to the two static program characteristics Genesis programs (i.e., the original and modified ones in Figure 6.5 and Figures 6.7/6.8), and ILGP and ILGP2 refer to the two image layering Genesis programs (i.e., the original and modified ones in Figures 6.9 and 6.11). The graph shows that runtime scales linearly with the number of generated instance programs in all case studies. Given that the image filtering and static program characteristics case studies contain nested `genloop` constructs,
while the image layering Genesis programs contain a single `genloop` construct with fewer Genesis entities, the time to generate programs for the former is an order of magnitude higher than the time for the latter. Finally, even for large numbers of generated instance programs, the time remains in the tens of minutes, leading us to conclude that the time for generating programs is reasonable.

The time to generate programs is broken down into three components: reading and parsing the Genesis program, generating instance programs, and writing instance programs to files. This breakdown is shown in Table 8.3 for the image filtering case study. Reading the Genesis program is done once for each invocation of the preprocessor, and thus the runtime for this phase remains constant and almost negligible. The other two phases grow linearly as the number of programs generated increases and constitute the bulk of the runtime with the instance program generation component dominating.
Chapter 9

Related Work

In this chapter, we discuss related work to Genesis. Section 9.1 describes existing code generators. Section describes existing testcase generators. Section 9.3 discuss preprocessors. Section 9.4 discusses previous work on describing languages. Section 9.5 looks at other methods of obtaining training data for machine learning.

9.1 Program Generators

Current program generators are used for many reasons. The University of Utah developed CSmith, a tool used to generate random C programs and is used to find bugs in compilers through stress testing [55]. CSmith generate multiple programs like Genesis. However, the programs are not fully described by the user. Command line arguments allow for some diversity of the programs it generates (e.g., allowing arrays), but programs are generally random to allow for stress testing.

CodeSmith Generator [14] is a commercially available product that can generate Visual Basic code using templates similar to Genesis. Generally, CodeSmith Generator is used to minimize repetitive tasks. The user of CodeSmith Generator uses tables to keep values which can be accessed in a repetitive manner. For example, repeatedly writing the values in the table can be automated using CodeSmith loops. However, CodeSmith Generator
does not provide sampling like Genesis, and consequently, does not generate multiple similar versions of a program with different characteristics.

TestMake [37] was developed by the Florida League of International Baccalaureate Schools (FLIBS), and is used to generate test harnesses for programs. The user indicates a piece of code, and programs are generated around it. Genesis allows whole programs to be described by the user. TestMake only considers the input and output of a piece of code and generates programs to use the code.

Bazzichi et al. [6] created an automatic generator for compiler testing. The generator produces a set of programs covering the grammatical constructions of a context-free grammar language. This limits the amount of languages it supports. It also does not give the user control over the programs generated beyond selecting a random seed.

Voronenko et al. [52] automated the generation of vectorized and multi-threaded linear transform libraries, providing users with optimized code for this domain of applications. Milder et al. [38] also extended this to automate the generation of linear digital signal processing transforms on hardware. In contrast, Genesis offers the flexibility to express programs in any domain, but does not optimize them, although optimizations can be manually expressed with Genesis.

Kamin et al. [32] created Jumbo, which generates code for Java during the actual running of the program. Poletto [44] has also added language and compiler support to generate code during runtime. In contrast, Genesis generates code, but does it during compilation and not runtime. Genesis also generates multiple programs when it is run taken from statistical samples instead of runtime information.

9.2 Testcase Generators

Generators can also generate testcases in forms other than programs. UDITA [25] is another expressive language and generator used to create testcases, but is used specifically
for Java programs. A user writes a test generation program in the UDITA language. The implementation of UDITA then reads the code to generate a large number of testcases to avoid manual generation of individual tests. Thus, Genesis and UDITA have many similarities in the process of using the generators. In contrast, Genesis is used to generate test programs and not test inputs for programs. Genesis is also more flexible and used with any language.

*AutoRT* [45] automatically generates parallel unit tests to detect data races in programs. Genesis’ intended use is for generating programs for use as testcases, but it can be used to generate test programs for multiprocessors, but external help is needed for it to be used to detect data races. Other test generators were used in the field for functional processor verification [17, 16, 2, 31, 1].

### 9.3 Preprocessors

Our work relates to approaches that customize programs, including work on preprocessors. A preprocessor is a program that takes in some input to produce code that is preprocessed for input into a processor [21]. Preprocessors can be classified into two types: *lexical preprocessors* [50] and *syntactic preprocessors* [34, 48]. Lexical preprocessors concentrate on the substitution of code, while syntactic preprocessors add syntax and functionality to an existing language. Preprocessors usually read inputs written in a preprocessor language.

An example of a lexical preprocessor is the C Preprocessor [21]. It is most often used to substitute code, particularly with the \#include directive. The C preprocessor also contains conditional substitutions using \#ifdef and \#define directives. Genesis provides substitution in a similar manner, but can be used to substitute in partial lines, while the C preprocessor is used for entire lines of codes. Also Genesis provides the use of loops without using recursion or Boost [4].
Another lexical preprocessor language is m4. It allows the user to define lines with arguments, and use these lines in the output [33], similar to the idea of features in Genesis. However, it does not provide some of the automatic options that Genesis provides, such as varlist generation, as it makes no assumptions on the type of code it generates. Also, unlike these lexical preprocessors, Genesis allows for the sampling of values, allowing for multiple generated instance programs.

A second type of preprocessor is the syntactic preprocessor. These preprocessors are generally used to change or extend the syntax of a language [34]. These syntax extensions are described and read by a syntactic preprocessor, which converts any use of these extensions to a format that can be read by the later compiler stage. Examples of languages for syntactic preprocessors to extend their own languages are Lisp [34] and OCaml [48]. While Genesis also converts lines that cannot be read by the compiler and converts the code to a form readable by a later compiler stage, Genesis is different than this type of preprocessor as it does not extend the language of the synthetic programs generated.

9.4 Describing Programs

Our work also relates to the Program Description Language [11] another approach that describe languages. Program Description Language (PDL, sometimes called Program Design Language) is a language used in software design to describe code developed by Caine et al [11]. PDL is a form of pseudocode, where programs are described in simple English (e.g., Turn on LED). While PDL is used to describe single programs, Genesis is used express larger program spaces.

Leung et al. [35] uses hypergraphs as a modelling language to describe inputs to programs to generate testcases. While the Genesis language is a form of modelling inputs to the preprocessor, Leung et al. are modelling inputs in a program differently in the
traditional sense, as inputs to a program. Genesis is more accurately classified as a program generation language.

### 9.5 Training Data for Machine Learning

There exist repositories of training data available to researchers for many application domains. For example, the UC Irvine Machine Learning Repository contains over 200 data sets for use by the ML community [51]. This approach to collecting training data is different than Genesis, which generates the actual programs which the data is collected from.

CTuning’s Collective Optimization Database (cDatabase) [18] aims to also provide sufficient training data for machine learning in the domain of program performance auto-tuning. It does so through a repository of optimization and performance data for real programs gathered from cTuning users. In contrast, Genesis provides users with more control over the definition of program characteristics and the generation of diverse training programs, albeit synthetic ones.
Chapter 10

Conclusions

In this thesis, we presented Genesis, a language to express and generate statistically controlled program sets for use in multiple domains and applications. It is different from previous preprocessors by providing the unique ability to sample from distributions. It is not restricted to a specific output language and is also flexible enough to express sets of programs with varying lengths and characteristics. We presented three case studies in different domains to illustrate the utility of Genesis and its ability to easily express programs with different characteristics. We designed and implemented a prototype preprocessor for Genesis. We evaluated the preprocessor’s performance and demonstrated the statistical quality of the samples it generates.

This work can be extended in several directions. More case studies can be used to assess if the Genesis constructs should be extended to increase functionality or usability. For example, the amount of options for Genesis values can be extended. One option is adding partial enumeration functionality for a Genesis value, where a user can indicate the number of values to enumerate through instead of enumerating the entire distribution, giving a balance between random sampling and full enumeration. Similarly, more control can be given to hold values for a specific number of generated instance programs. These functionalities are currently possible with smart code or workarounds, but the ease of
using Genesis may increase from adding specific constructs for these options.

Many programming languages that accept arguments to functions also allow functions to return values [27, 40]. This is analogous to a Genesis feature returning values to the higher-level feature that references it. While this functionality is not needed for the current case studies, it is a possible extension to Genesis' functionality in the future.

Finer typing can be added to Genesis. Genesis entities have typing in the form of values and variables, but the references only result in string values, or a random real value based on the implementation of Perl. Genesis values may possibly be extended to distinguish between numbers and strings, add stronger typing, or possibly add an Uncertain type [10].

Future work can also be in the direction of improving the functionality and efficiency of the Genesis preprocessor, improving performance and reducing memory footprint. The design of Genesis makes it suitable for parallelization, which will be a greatly beneficial improvement.

A third direction involves adding target language-specific improvements to Genesis. For convenience, proper highlighting can be added to Genesis constructs when working on code editors like Emacs and Vim. This may be complicated as a different highlighting scheme is needed for each possible target program language, which will be difficult to remain sustainable.

Also, if the instance programs being generated are known to be written in OpenCL, it is possible to generate the host program to allow the user to run the programs and get runtime information directly after using Genesis. This increases the user-friendliness as it removes the need for the user to write their own host program, but comes with the limitation of requiring target language-specific knowledge, which goes against one of the goals of Genesis.

We believe that Genesis is a useful tool that eases the expression and creation of large and diverse program sets, which can provide large benefits for its users. Genesis has
been used by others to generate stencil programs for research in GPU optimization [24], showing that Genesis has made an impact in one of the domains that has motivated its creation. Indeed, synthetic programs such as those generated by Genesis have been shown to improve the quality of machine learning models [28].

We foresee Genesis as being useful in domains other than those discussed in this thesis and that also require synthetic programs or testcases. For example, testcases containing loop dependencies with various properties can be generated using Genesis to test compiler-based dependence detectors. Similarly, Genesis can be used to generate graphs with varying node and edge properties as inputs to test graph-based applications. To these ends, we have released the Genesis preprocessor into the public domain as an open source software artifact\(^1\).

\(^1\)https://github.com/chiualto/genesis
Appendix A

Full Preprocessor Specification

The following is a description of the Perl Genesis Preprocessor, current for Version 1.01.300.

Appendix A.1 explains how to use Genesis using the command line. Appendix A.2 goes over the constructs that Genesis allows. Appendix A.3 lists the possible errors and warning messages of Genesis.

A.1 Running in the Command Line

Assuming that the Genesis preprocessor implementation genesis.pl is in the present working directory, in the general case, Genesis can be run by using:

./genesis.pl [Genesis file]

with the Genesis file containing both the Genesis program and the template program underneath. Perl may have to be explicitly declared, in which case perl is added in front of the previous command (or any other commands) as follows:

perl ./genesis.pl [Genesis file]

Alternatively, if the Genesis program and template program are in separate files:

./genesis.pl [Genesis program] [template program]

Inputting ./genesis.pl -h outputs a usage line:
Usage: ./genesis.pl [Genesis file] [options]
Usage: ./genesis.pl [Genesis program] [template program] [options]

Possible options:
dataoutfile (can be Any Char-string)
outfile (can be string|string (ex. Gen*.out))
outdir (can be Any Char-string)
ignorevertspace (default: 0. 1)
generate (default: 1. 0)
printP (default: 2, 1, 3, 0)
printG (default: 2, 1, 3, 0)
globalcounters (default: 0. 1)
chisquared (default: 0. 1)
header (default: 1. 0)
headercomments (default: 0. 1)
recursion (default: 1. 0)

Example usage:
./genesis.pl ./GenesisPrograms/Testcases/spec-demo.c printP=2 printG=2

Table A.1 shows all the options available in the current preprocessor.

A.2 Constructs

Section A.2.1 gives a brief overview of the constructs that Genesis allows. This section goes in-depth on each construct.
When describing the format of each construct, non-italicised words are required when the construct is used, while italicised words provide options that are not required. Bolded words are constructs, written as shown, while non-bold words can be a string of alphanumeric characters and underscores or numbers, based on context explained in the section.

A.2.1 Constructs Overview

The following are a quick view list of constructs in Genesis.

begin genesis
Starts the genesis program.
distribution
Defines global distributions outside of the generate statement.
value
<table>
<thead>
<tr>
<th>Argument #</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genesis program</td>
<td>A string</td>
<td>The Genesis program to be parsed.</td>
</tr>
<tr>
<td>outfile</td>
<td>String*string</td>
<td>The instance program format for instance programs</td>
</tr>
<tr>
<td></td>
<td>Ex. gen*.out</td>
<td></td>
</tr>
<tr>
<td>dataoutfile</td>
<td>A string</td>
<td>A place to put the parsing info if to the terminal.</td>
</tr>
<tr>
<td>finaloutfile</td>
<td>A string</td>
<td>The file for the final summary (not terminal).</td>
</tr>
<tr>
<td>outdir</td>
<td>A string</td>
<td>The directory for instance programs.</td>
</tr>
<tr>
<td>ignorevertspace</td>
<td>0</td>
<td>Does not ignore vertical spacing.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Ignores blank lines. Use “genspace” to force.</td>
</tr>
<tr>
<td>generate</td>
<td>0</td>
<td>Parse Genesis program, do not generate.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Parse Genesis program, generate instance programs.</td>
</tr>
<tr>
<td>printP</td>
<td>0</td>
<td>No info while parsing the Genesis program.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Show a summary of parsed info at the end.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Show info while parsing the Genesis program.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Show more info while parsing the Genesis program.</td>
</tr>
<tr>
<td>printG</td>
<td>0</td>
<td>No info while generating instance programs.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Show a summary of the instance programs at the end.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Show info while generating instance programs.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Show more info while generating instance programs.</td>
</tr>
<tr>
<td>globalcounters</td>
<td>0</td>
<td>Do not print the global counters.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Print the global counters.</td>
</tr>
<tr>
<td>chisquared</td>
<td>0</td>
<td>Do not output the chi-squared test.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Output the chi-squared test.</td>
</tr>
<tr>
<td>header</td>
<td>0</td>
<td>Do not print a header in each instance program.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Print a header in each instance program.</td>
</tr>
<tr>
<td>headercomments</td>
<td>0</td>
<td>Do not treat lines outside sections as comments.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Treat lines outside sections as comments.</td>
</tr>
<tr>
<td>recursion</td>
<td>0</td>
<td>Do not allow recursion.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Allow feature recursion.</td>
</tr>
</tbody>
</table>

Table A.1: Table of Command Line Arguments
Defines a Genesis value that is sampled for each instance.

**varlist**
Defines a pool of variables, which Genesis variables can be sampled from.

**variable**
Defines a Genesis variable that is sampled for each instance.

**feature**
Defines a feature, defining a code snippet and how it varies.

**genif**
Defines an if statement that can generate code based on a sampled or defined value.

**genloop**
Defines a loop containing a iterator value that can be used in the loop. Alternatively, Defines a loop with a condition.

**add**
Removes from a varlist.

**remove**
Adds to a varlist.

**genmath**
Evaluates the code in the brackets, or forces evaluation.

**genassert**
Asserts that a boolean expression is true. Ends the instance program if false.

**/// or gencom**
Defines a comment that is not put into the actual code.

**generate**
Determines how many to generate and defines the distributions.

**end genesis**
Ends the Genesis program.

### A.2.2 begin genesis Construct

This construct is needed at the beginning of the Genesis program and included in every Genesis program.

**Format**

```
begin genesis
```

The line is the same in every Genesis program.

### A.2.3 distribution Construct

The **distribution** construct defines a set of values and their corresponding probabilities, storing these under a Genesis name. Distributions can either be defined using this statement or with the generate construct. Distributions are set once declared. This means that distributions declared using Genesis values will not change if the Genesis value is changed later on in processing.
Format

distribution genesisName = distributionRange
All the distributions are declared here.

distributionRange format is the following:
{value:endValue{distribution or increment};
value:endValue{distribution or increment}... ;;distributionType}
Distribution percentage can be math. If distributions do not add to 100, the distributions
are normalized, but the distributions still must be originally between 99 and 101 (or 0.99
and 1.01). Distributions can be a number (e.g. 0.5), a percentage (e.g. 50%) or a math
expression (e.g. 1/2 or 4/2-3/2). Not including a distribution type defaults to uniform
random. Currently, Genesis only allows uniform distributions, but other distribution
types (e.g. normal distribution) can possibly be implemented in the future. Increment is
better shown through an example (see examples dist4, dist5, and dist6).

Examples

distribution dist0 = {1:10}
Any Genesis value using dist0 as its distribution has a 10% chance of sampling each
value from 1 to 10.

distribution dist1 = {0{1/4}, 1{1/8}, 2{0.375}, 3:10{1/4} }
The chances of each value of being sampled are: 0 has a 1/4 chance, 1 has a 1/8 chance,
2 has a 3/8 chance, 4-10 have a 1/4, uniform for each value inside.

distribution dist2 = {0, 1, 4:10;;uniform }
0, 1 and 4-10 have a 33 chance each, individually, each value from 4-10 has a 33/7 chance
each.

distribution dist3 = {0, 1, 4:10}
dist3 is same as dist2 as default distribution is uniform.

distribution dist4 = {2:32{*2}}
dist4 shows an increment in the brackets. Here the values in the distribution are 2, 4, 8,
16, 32, each with a 20% chance.

distribution dist5 = {2:32{+2}}
dist5 shows an increment in the brackets. Here the values in the distribution are 2, 4, 6,
8,..30, 32.

distribution dist6 = {0{40}, 2:32{*2, 60}}
dist6 contains 0 with a 40% change, and {2,4,8,16,32} with a 60% chance (so 2 has a 12% chance,
4 has a 12% chance, etc.).
distribution dist7 = { 4:10;; real }
All real values from 4-10 can be sampled.

Restrictions
Using real assumes a single range only.
An error message occurs if more than 1 probability or increment is attached to a value.
An error also occurs of the probabilities do not add up to 100, or a value is missing a probability when the other values in the distribution have probabilities.

A.2.4 value Construct
The value construct defines a Genesis entity whose value is sampled from a distribution.
Values can be used with other constructs or can be propagated as a constant to the instance program. It can either take on a distribution, or copy the same value of a previously defined Genesis value.

Sampling Format

value genesisName sample takenDistName modifiers
This format is used for a value to be sampled from a distribution. genesisName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenDistName can be a string a characters, but should exist as a distribution declared earlier in a distribution statement, or later with the generate statement. genesisName, when sampled, takes on a value based on the distribution takenDistName.

value genesisName sample {distribution} modifiers
A distribution can be declared in-line.

Enumerate Format

value genesisName enumerate takenDistName modifiers
This format is the same as above, but using the word enumerate instead of sample. Instead of sampling from a distribution, a Genesis value can also enumerate one. This causes genesisName to enumerate through all the values in takenDistName, taking on every possible value of takenDistName, one per instance program.

Math Format

value genesisName = ${takenGenesisName} modifiers
value genesisName = mathExpression modifiers
These formats are used when a previously sampled value, or a math expression is used to calculate the new value. genesisName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenGenesisName can be a string of characters, but should exist as a Genesis value declared previously. mathExpression
is a math expression; all sampled values (if any) is replaced into the formula and the resultant value is stored as the value.

Initialization Format

value genesisName
This initializes a Genesis value, which can be assigned a sampled value later, using genmath. Its use without a value results in an error. This can be useful with multiple genif statements; initializing the value to set its declaration location, and using genif statements to assign it a value.

Other Formatting Details
Values declared over multiple lines can be compressed to a single line by separating declared values with commas:

value genesisName,genesisName2... sample takenDistName, genesisName3,genesisName4... sample takenDistName2, ...
Here, multiple values are declared using the same distribution, with multiple declarations on a single line. In general, these values are sampled with replacement.

Alternatively, arrays can be used for multiple sampling.

value genesisName[number] sample takenDistName
This results in number of genesisNames, referred in the code as genesisName[1], genesisName[2]... genesisName[number].

Modifiers
Modifiers can be added onto the end of lines, with default values used if not indicated. Values have a few adjustable modifiers. Setting an modifier twice results in an error.

report
With reporting set to 1 report(1), each time the value is sampled and used, a line is added at the beginning of each instance program indicating its name and its sampled value. The header command line option has to be turned on. The code results in the following at the top of each file:

//Sampled Reported Values:
//Set value numEpochs: 1
// Set value numComps: 5
// Set variable dest: temp5
noreplacement
With this modifier set to 1, multiple values sampled in the current line are sampled without replacement.
Examples

\begin{verbatim}
value stride sample dist1 report(1)
A Genesis value is declared called stride and it samples a value from a distribution called dist1. By using report(1), each time it is sampled, the result is reported as a comment at the beginning of each instance program.

value stride2 = ${stride}
A Genesis value is declared called stride2 and it has the same value sampled by stride.

value stride3 = ${stride}+1
A Genesis value is declared called stride3 and it has the same value sampled by stride plus 1. This value does not have to be part of the original distribution. If the distribution used by stride is 1:10, and stride is sampled to be 10, stride3 is 11.

value stride4 enumerate dist1
A Genesis value is declared called stride4, which enumerates through all values in dist1, one per instance program.

value stride5
A Genesis value is declared called stride5, without a given or sampled value. A value can be given later using genmath.

genmath stride6 sample {1:10}
stride6 is sampled from the declared distribution.
\end{verbatim}

Sampling without replacement

More than one value can be sampled at the same time using commas. By adding without replacement, the 2nd sampled value samples from the remaining variables not already sampled.

\begin{verbatim}
value swapIter,swapIter2 sample swapIterDist without replacement
swapIter samples from swapIterDist. swapIter2 samples from the remaining values in swapIterDist.
\end{verbatim}

An error is given if the sampling results in an empty pool. For example, if 5 values are to be sampled without replacement when the distribution contains 4 numbers, this results in an error.

Arrays can be dynamic, as of Genesis 1.01.000. For example, the following code is now valid:

\begin{verbatim}
value test sample dist1 report(1)
value arrayEx[${test}] sample dist1 report(1)
A dynamic value array of size test is created.
\end{verbatim}
Limitations

Enumerate is not properly implemented when used inside a feature. Enumerate only works on the regular level of the global and program section. Using `enumerate` constructs in these sections may not work as intended.

A.2.5 `varlist` Construct

The `varlist` construct defines a pool of variables that can be used in the processed feature and hence, be part of the instance program. The section in which the varlist is declared indicates the reinitialization rate of the varlist. If a varlist was manipulated using `add` or `remove` constructs, a varlist is *reinitialized* by undoing those actions, returning to a full varlist with all its variables. Varlists declared in the global section are created once for the entire set of programs. The size of the varlist and state of variables are maintained between instance programs in this case. Varlists declared in the program section are reinitialized at the point of its declaration, and thus, it return to a full varlist with all its variables for each instance program. Varlists declared in a feature are local to that feature only and are reinitialized for each processing of the feature.

Format

```
varlist varlistName [number] modifiers
varlist varlistName [genesisName] modifiers
```

`varlistName` can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. `number` indicates a set number of variables; `genesisName` allows for a variable number of variables based on a Genesis value that was declared earlier.

Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Varlists have a few adjustable modifiers. Setting an modifier twice results in an error.

```
name
```

`name` allows the user to define a character string that goes before the number in the output variables. By default, if this modifier is not used, the name of the varlist is used as that string. For example, a varlist declared using `temp[5]`, has variables ranging from `temp1` to `temp5`. However, if it is declared using `temp[5] name(foo)`, the variables are named from `foo1` to `foo5`.

Examples

```
varlist vars[5]
```

A `varlist` called `vars` containing 5 variables is initialized, named from `vars1` to `vars5`.

```
varlist foo[${someValue}] name(bar)
```
A varlist called `foo` containing some variables, where the number of variables is `someValue`, a value that should be declared as a Genesis value, and sampled in each instance of the program. The variables in the list are named `bar1`, `bar2`, etc.

**Varlist References**

When a varlist is referenced, statistics of the varlist can be queried using arguments.

`(size)`

size returns the size of the current varlist. For example, declaring using `temp[5] name(foo)` and referencing using `${temp(size)}` outputs 5, in the general case. Using adds and removes can affect its return value.

`(name)`

name returns the name used in the variable. For example, declaring using `temp[5] name(foo)` and referencing using `${temp(name)}` outputs `foo`.

`[a number]`

Referencing with a number between the square brackets accesses that variable in the varlist. For example, declaring using `temp[5] name(foo)` and referencing using `${temp[3]}` outputs `foo3`.

**Varlist Reference Examples**

```plaintext
varlist bar[10]
genif ${bar} > 5
end
genif ${bar} > 15
end
```

The first `genif` evaluates to true and the second evaluates to false. Using add and remove constructs can manipulate the evaluation of `bar`.

```plaintext
value stride1 sample a_dist
varlist my_vars[5] name(foo)
print "${temp(size)}";
print "${temp(name)}";
print "${temp(4)}";
```

The first varlist reference outputs 5, the varlist’s size. The second reference outputs `foo`, the name used in all the variables in the varlist. The third reference outputs `foo4`, the specific name of the 4th variable in the varlist.
A.2.6 variable Construct

The variable construct defines a Genesis entity whose value is sampled from a varlist and is propagated as a variable name to the target program. It samples from a variable pool with uniform distribution, or takes on the same variable name as another Genesis variable.

Each Genesis variable keeps track of their pool of available variables that can be pooled. These lists can be modified using the add and remove constructs.

Format

```
variable variableName from takenvarlistName modifiers
```

variableName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenvarlistName can be a string of characters, but should exist as a varlist declared earlier in the code. variableName, when sampled, takes on a variable in that varlist.

```
variable variableName = takenVariableName modifiers
```

variableName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenVariableName can be a string of characters, but should exist as a variable declared previously.

Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Variables have a few adjustable modifiers. Setting a modifier twice results in an error.

```
report
```

With reporting set to 1 `report(1)`, each time the value is sampled and used, a line is added at the beginning of each instance program indicating its name and its sampled value. The header command line option has to be turned on. The code results in the following at the top of each file:

```
//Sampled Reported Values:
//Set value numEpochs: 1
// Set value numComps: 5
// Set variable dest: temp5
```

Examples

```
variable tempVar from vars
```

Variable tempVar is sampled from a varlist vars.

```
variable tempVar2 = ${tempVar}
```

Variable tempVar2 takes on the same variable as tempVar.
variable tempVar3 = \${tempVar}+1
Variable tempVar3 takes on the variable value after tempVar. For example, if tempVar is var1, tempVar3 takes on the variable var2. This DOES wrap around, as variables depend on factors such as initialization, that values do not require.

A.2.7 genmath Construct
The genmath construct allows the re-evaluation of previously declared Genesis values.

Math Format
```
genmath genesisName = someExpression
```
These formats are used when a previously sampled value, or a math expression is used to calculate the new value. genesisName and takenGenesisName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character, as long as the Genesis name exist as a Genesis value declared previously. mathExpression means that math can be used; all sampled values (if any) is replaced into the formula and the resultant value is stored as the value.

Sampling Format
```
genmath genesisName sample takenDistName modifiers
```
This format is used for a value to be sampled from a distribution, useful for when a value is declared but not yet sampled. genesisName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character, as long as it exists as a Genesis value declared previously. takenDistName can be a string a characters, but should exist as a distribution declared earlier in a distribution statement, or later with the generate statement. genesisName, when sampled, takes on a value based on the distribution takenDistName.

```
genmath genesisName sample \{distribution\} modifiers
```
A distribution can be declared in-line.

Examples
```
genmath stride sample dist1
A Genesis value is declared called stride and it samples a value from a distribution called dist1.

ngenmath stride2 = \${stride}+1
A Genesis value previously declared called stride3 takes the same value sampled by stride plus 1. This value does not have to be part of the original distribution. If the distribution used by stride is 1:10, and stride is sampled to be 10, stride2 is 11. The idea is that stride3 should not be held down by the restrictions of stride, since a
distribution is not indicated. There may be an argument in a future update that may control this.

```
genmath stride3 sample {1:10}
```

stride3 is sampled from the declared distribution.

### A.2.8 add Construct

The **add** construct adds a variable to a varlist or a value to a distribution to affect future samplings.

**Format**

```
add variableName to destinationName
```

*variableName* can be any string of alphanumeric characters and underscores, either the name of a value, variable, or a number. *destinationName* can be a string a characters, but should exist as a varlist or distribution declared earlier in the code. The value of *variableName* is added to the allowable sampled values of *destinationName*.

**Examples**

```
add dest to vars
```

Add the sampled value of *dest* into the *vars* varlist.

```
add stride to bar
```

Add the sampled value of *stride* into the *bar* varlist.

```
add 1 to bar
```

Add the first variable back into the *bar* pool.

```
add 1 to dist1
```

Add 1 into *dist1*. It is normalized with the rest of the values in the distribution.

### A.2.9 remove Construct

The **remove** construct removes a variable to a varlist to affect future samplings, such that it can be used to prevent repeated samplings.

**Format**

```
remove variableName from sourceName
```

*variableName* can be any string of alphanumeric characters and underscores, either the name of a value, variable, or a number. *sourceName* can be a string a characters, but
should exist as a varlist or distribution declared earlier in the code. The sampled value of variableName is removed from the allowable sampled values of sourceName.

Examples

- `remove dest to vars`
  Removes the sampled value of dest from the vars varlist.

- `remove stride to bar`
  Removes the sampled value of stride from the bar varlist.

- `remove 1 to bar`
  Remove the first variable from the bar pool.

- `remove 1 to dist1`
  Removes 1 from dist1, assuming it exists in the distribution. The remaining values in the distribution are normalized.

A.2.10 feature Construct

The feature construct defines a code snippet built up using Genesis names or possibly other features. A feature is defined inside its own block in the feature section of the Genesis program.

Format

```
feature modifiers featureName
  code
end
```

featureName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. Description can be a set of code in the output language or declared local values and variables, or even other features.

```
feature modifiers featureName(listOfArguments)
  code
end
```

Arguments can be used to pass values into the code. The code can use these arguments as if the arguments were Genesis values.

Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Features have a few settable modifiers.
**singleline**
Default value is 0. Setting it to 1 results in the resultant code snippet to be in a single line with no newlines, and thus, the resultant C line is in a single line. For example, it can be used if a feature contains a `genloop` iterating through a varlist.

**Examples**

```plaintext
feature access
  variable dest from temp
  value stride1 sample dist1
  value stride2 sample dist1
  value offset sample dist1
  ${dest} = arr[${stride1}*it00 + ${stride2}*it01 + ${offset}];
end
```

This declares a feature named `access`. Values and variables sampled from distributions and varlists, and are used in the code snippet.

```plaintext
feature computation
  variable dest from temp
  variable source1 from temp
  variable source2 from temp
  variable source3 from temp
  ${dest} = ${source1} * ${source2} + ${source3};
end
```

A computation is generated and the code snippet replaces that feature reference.

```plaintext
feature epoch
  value numComps sample compDist
  ${computation[${numComps}]}${access}
end
```

Example of using other features in the code snippet. Here, features are built using other features.

```plaintext
feature access2 (stride1,stride2,offset)
  variable dest from temp
  ${dest} = arr[${stride1}*it00 + ${stride2}*it01 + ${offset}];
end
```

This example takes 3 arguments used in the code itself. See feature usage for examples on how to use this.

**Restrictions**

Feature names cannot contain take the same name as another Genesis entity.
Values declared in a feature can only be used in that feature. For use in a sub-feature, pass it as an argument.

### A.2.11 feature Usage

A feature is used in the template program or in another feature by its name surrounded by curly brackets after a dollar sign, either in the template program or in other feature definitions. For each feature reference, the feature is processed and the resultant code snippet is substituted into the feature reference.

Features can also have arguments passed in by value.

#### Format

```
featureName (arguments) [number or genesisName]
```

A feature is referenced feature name as declared in the program.

#### Optional Clauses

Brackets allow arguments passed in by value, as required by the definition. Using square brackets allow for repetition. Inside the bracket can be a number or a Genesis value.

#### Examples

```
${access}
```

A single access replaced at the location of the reference.

```
${computation[5]}
```

5 differently sampled computations replaced at the location of the reference.

```
value numEpochs sample {1-5}
${epoch[$numEpochs]}
```

Generate numEpochs number of epochs to replace this reference.

```
${access2(1,2,3)}
```

Here, access2 is used with 1, 2, and 3 passed as arguments. Using the above example of the definition of access2, the arguments correspond to stride1, stride2 and offset respectively.

```
${access2(1,2,3)[5]}
```

This example shows the ordering of arguments and square brackets when used together.
### A.2.12 Stored features

Processed features can also be stored for later referencing by a given Genesis name. A reference with the given Genesis name is substituted by the already processed code without reprocessing, similar to a Genesis value or variable.

**Format**

```
feature featureName process takenFeatureName modifiers
```

featureName can be any string of alphanumeric characters and underscores, either the name of a value, variable, or a number. takenFeatureName can be a string a characters, but should exist as a feature declared in the code. An instance of takenFeatureName is processed and stored into featureName. It can then be used and replaced by the same previously processed feature instance.

**Examples**

```
feature stored_computation process computation
```

A stored feature is declared called stored_computation as a stored processing of computation. The feature is processed during this line, and then can be used as many times as needed.

### A.2.13 genif Construct

The genif construct is used for conditional generation of code snippets.

**Format**

```
genif booleanExpression
description1
end
```

If booleanExpression is determined to be true, then description1 is processed and placed into the instance program. Otherwise, this section of the Genesis program produces no code.

**Optional Clauses**

Using genelsif constructs after a genif statement allow for a second condition block that is only evaluated if the first genif statement is evaluated to false. genelse constructs allow a code section to be processed if all preceding genif and genelsif statements were evaluated to be false.
If $booleanExpression$ is determined to be true, then $description1$ is processed and placed into the instance program. Otherwise, it continues checking each boolean expression until it finds a true expression, in which that description is processed. In no true expression exists, it processes the $genelse$ clause. Even if no $genelse$ clause exists, the end of the block needs to include the construct $end$.

Examples

```plaintext
genif $ifChoice==1$
  code
end
genif $ifChoice==2$
  otherCode
end
```

ifChoice is a Genesis value. If it is sampled as 1, code is used. If it is sampled as 2, otherCode is used. Any other value and neither are used (this part remains blank in the actual instance). Both sections are checked; the genif statements are independent of each other.

```plaintext
genif $ifChoice==1$
  code
 genelsif $ifChoice==2
   otherCode
 genelse
   moreOtherCode
end
```

ifChoice is a Genesis value. If it is sampled as 1, code is used. If it is sampled as 2, otherCode is used. Any other value and neither are used (this part remains blank in the actual instance). Note here that if one section is evaluated to be true, further sections are not evaluated.
A.2.14 genloop Construct

The genloop construct facilitates repetitive code generation. The genloop construct can either be a counter format based on an increasing iterator, or can also test boolean conditions (similar to a while loop).

Counter Format

```
genloop genesisName:valueStart:valueEnd:valueStride
description
end
```

The description is repeated $valueEnd - valueStart + 1$ times. genesisName can be used as a Genesis value, which takes on the values from valueStart to valueEnd, and thus, can be used in the code.

Optional Clauses

A valueStride can be indicated. A stride of 1 is default.

Conditional Format

```
genloop condition
description
end
```

This format is to use a genloop based on a condition (similar to a while loop). It is used with varlist with add/remove used.

Examples

```
genloop loopVar:1:5
    ${access(${loopVar})}
end
```

loopVar can be used like a Genesis value. It takes on the value of 1, and can be used in the code. Then, in the next iteration of the genloop, it takes on the value of 2, and can be used in the code. In this example, the code is repeated 5 times. access, assuming it was defined with an argument, takes the loopVar as an argument, taking on a different value in each iteration.

```
genloop loopVar:1:5:2
    ${access(${loopVar})}
end
```

Here, this line is the same as above, but the stride is 2. Thus, 3 accesses are made, where 1, 3 and 5 are passed in.
A.2.15 genassert Construct

The **genassert** construct makes an assertion of a boolean expression, similar to a **genif** statement. If the expression is evaluated to be true, processing of the instance program continues normally. However, if it is false, processing stops for the current instance program. Instead, it is considered a failed program and is deleted. The preprocessor then continues processing the next instance program. For example,

**Format**

```
genassert booleanExpression
```

It has a similar format to a **genif** statement. If booleanExpression is evaluated to be true, then processing continues normally. If it is false, processing stops for the current instance program, and processing continues for the next instance program.

**Examples**

```
genassert ${ifChoice}==1
```

*ifChoice* is a Genesis value. If it is a value and sampled as 1, the generated instance program is valid at this point. If it is not 1, then the assumption is that something is wrong, and the current instance program has failed.

A.2.16 geneval Construct

This evaluates the code in the bracket and placed in the instance program after being evaluated.

**Format**

```
geneval(toBeEvaluated)
```

Here, toBeEvaluated is any evaluable string (i.e., a string containing numbers or the characters `+ - * /`), and is evaluated before being substituted into the instance program.

**Examples**

```
2+1
```

is translated directly into the instance programs, without being evaluated.

```
geneval(2+1)
```

results in 3 being put in the instance programs.

A.2.17 generate Construct

The **generate** construct defines how many instance programs to generate. The generate construct also allows the definition of global distributions.
Format

```
generate number
```

*number* is an integer to determine how many to generate.

```
generate number with distribution,distribution,...
```

*number* is an integer to determine how many to generate. Distributions not yet declared can be declared here too, in the same format as distributions declared using the *distribution* construct.

Examples

```
generate 500
```

Most basic generate statement with no distributions declared.

```
generate 500 with dist1 = {0{1/4}, 1{1/8}, 2{0.375}, 3:10{1/4} }
```

The chances of each value of being sampled are: 0 has a 1/4 chance, 1 has a 1/8 chance, 2 has a 3/8 chance, 4-10 have a 1/4, uniform for each value inside.

```
genenerate 100 with dist2 = {0; 1; 4:10;; uniform}
```

0, 1 and 4-10 have a 33 chance each, individual 4-10 have a 33/7 chance each.

```
genenerate 200 with dist3 = {0; 1; 4:10} , dist4={1;2}
```

dist3 is same as dist2 as default distribution is uniform. This example also shows multiple distributions being declared at once.

```
genenerate 100 with dist5 = { 4:10;; real }
```

All real values from 4-10 can be sampled.

```
genenerate 100 with dist6 = { 1:16{*2}}
```

dist6 shows an increment in the brackets. The values of powers of 2 from 1-16 (1,2,4,8,16) are in the distribution.

Multiple generate lines in the same Genesis program is allowed.

```
genenerate 100 with dist7 = {4:10}
genenerate 100 with dist7 = {14:20}
```

This generates 200 total instance programs, 100 with each generate line.

Generate can be done over multiple lines, with an *end*. This keeps code cleaner.

```
genenerate 100 with
    dist8 = {4:10}
    dist9 = {0;1;15}
end
```
Restrictions

Using real assumes a single range only.

A.2.18 end genesis Construct

This construct is needed at the end of the Genesis program. The line is the same in every Genesis program.

Format

end genesis

The line is the same in every Genesis program.

A.3 Error and Warning Messages

Table A.2 contains a list of errors that can appear while processing with the Genesis Perl preprocessor.

<table>
<thead>
<tr>
<th>Code #</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>Invalid value line.</td>
</tr>
<tr>
<td>'B'</td>
<td>Number to be distributed invalid.</td>
</tr>
<tr>
<td>'C'</td>
<td>Invalid varlist line.</td>
</tr>
<tr>
<td>'D'</td>
<td>Invalid variable line.</td>
</tr>
<tr>
<td>'E/E1'</td>
<td>Invalid add line.</td>
</tr>
<tr>
<td>'F/F1'</td>
<td>Invalid rem line.</td>
</tr>
<tr>
<td>'G'</td>
<td>Feature repetition value/number not valid.</td>
</tr>
<tr>
<td>'H'</td>
<td>Distribution for this value does not exist.</td>
</tr>
<tr>
<td>'I'</td>
<td>Varlist for this variable does not exist.</td>
</tr>
<tr>
<td>'J'</td>
<td>Distribution range end less than start value.</td>
</tr>
<tr>
<td>'K'</td>
<td>Varlist line argument invalid.</td>
</tr>
<tr>
<td>'L'</td>
<td>Feature line argument invalid.</td>
</tr>
<tr>
<td>'M'</td>
<td>Weird distribution value. Right chars, but possibly wrong format.</td>
</tr>
<tr>
<td>'N'</td>
<td>Invalid feature line.</td>
</tr>
<tr>
<td>'O'</td>
<td>Weird distribution value. Some char not allowed.</td>
</tr>
<tr>
<td>'P'</td>
<td>Varlist line argument structure invalid.</td>
</tr>
<tr>
<td>'Q'</td>
<td>Nothing in the brackets for value array.</td>
</tr>
<tr>
<td>'R'</td>
<td>Number for value array is invalid.</td>
</tr>
<tr>
<td>'S'</td>
<td>Too many arguments in this feature.</td>
</tr>
<tr>
<td>'T'</td>
<td>Too few arguments in this feature.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Code #</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>Genmath string not valid.</td>
</tr>
<tr>
<td>'V'</td>
<td>RHS or expression not fully evaluated for Genmath.</td>
</tr>
<tr>
<td>'W'</td>
<td>LHS does not exist to genmath.</td>
</tr>
<tr>
<td>'X/X2/X3'</td>
<td>Duplicate name exists as a Value/Variable.</td>
</tr>
<tr>
<td>'Y'</td>
<td>Not enough distribution to sample all without replacement.</td>
</tr>
<tr>
<td>'Z'</td>
<td>Value line argument invalid.</td>
</tr>
<tr>
<td>'AA'</td>
<td>Value line argument structure invalid.</td>
</tr>
<tr>
<td>'AB'</td>
<td>Missing 'end' statement.</td>
</tr>
<tr>
<td>'AC'</td>
<td>No end genesis line.</td>
</tr>
<tr>
<td>'AD'</td>
<td>Way too many duplicates for some reason.</td>
</tr>
<tr>
<td>'AE'</td>
<td>Generate line in feature.</td>
</tr>
<tr>
<td>'AF'</td>
<td>Dynamic Not enough distribution to sample all without replacement.</td>
</tr>
<tr>
<td>'AG'</td>
<td>valueStart not fully evaluated.</td>
</tr>
<tr>
<td>'AH'</td>
<td>valueEnd not fully evaluated.</td>
</tr>
<tr>
<td>'AI'</td>
<td>valueStride not fully evaluated.</td>
</tr>
<tr>
<td>'AJ'</td>
<td>Genelsif without genif.</td>
</tr>
<tr>
<td>'AK'</td>
<td>Genelse without genif.</td>
</tr>
<tr>
<td>'AL'</td>
<td>Enumerate is invalid while in a feature.</td>
</tr>
<tr>
<td>'AQ'</td>
<td>Report argument more than once.</td>
</tr>
<tr>
<td>'AR'</td>
<td>Noreplacement argument more than once.</td>
</tr>
<tr>
<td>'AS'</td>
<td>Singleline argument more than once.</td>
</tr>
<tr>
<td>'AT'</td>
<td>Blank variable line.</td>
</tr>
<tr>
<td>'AU'</td>
<td>Start value not a valid value to sample from.</td>
</tr>
<tr>
<td>'AV'</td>
<td>End value not a valid value to sample from.</td>
</tr>
<tr>
<td>'AW'</td>
<td>Bad genif condition.</td>
</tr>
<tr>
<td>'AX'</td>
<td>Bad genelsif condition.</td>
</tr>
<tr>
<td>'AY'</td>
<td>Genelse should not have a condition.</td>
</tr>
<tr>
<td>'AZ'</td>
<td>Varlists is invalid (for now) while in a feature.</td>
</tr>
<tr>
<td>'BA'</td>
<td>Variable line argument invalid.</td>
</tr>
<tr>
<td>'BB'</td>
<td>Variable line argument structure invalid.</td>
</tr>
<tr>
<td>'BC'</td>
<td>Report argument more than once.</td>
</tr>
<tr>
<td>'BD'</td>
<td>Varlists using 'from' should not have any arguments.</td>
</tr>
<tr>
<td>'BE'</td>
<td>Weird argument for generate.</td>
</tr>
<tr>
<td>'BF'</td>
<td>Value in the brackets does not exist.</td>
</tr>
<tr>
<td>'BG'</td>
<td>End is less than start of range.</td>
</tr>
<tr>
<td>'BH'</td>
<td>Feature repetition value not allowed in template program.</td>
</tr>
<tr>
<td>'BI'</td>
<td>Underscores allowed but cannot be first char.</td>
</tr>
<tr>
<td>'BJ'</td>
<td>Probability is non-numeric.</td>
</tr>
<tr>
<td>'BK'</td>
<td>Start value not defined yet.</td>
</tr>
<tr>
<td>'BL'</td>
<td>End value not defined yet.</td>
</tr>
<tr>
<td>'BM'</td>
<td>More than one program section.</td>
</tr>
<tr>
<td>'BN'</td>
<td>Feature cannot be named program.</td>
</tr>
<tr>
<td>'BP'</td>
<td>Distribution used already.</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Code #</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BQ'</td>
<td>Add cannot work with real values.</td>
</tr>
<tr>
<td>'BR'</td>
<td>Remove cannot work with real values.</td>
</tr>
<tr>
<td>'BS'</td>
<td>Invalid value line. (Maybe change 'from' to 'sample'.)</td>
</tr>
<tr>
<td>'BT'</td>
<td>Not a valid varlist target for a variable.</td>
</tr>
<tr>
<td>'BU'</td>
<td>Too many probability arguments.</td>
</tr>
<tr>
<td>'BV'</td>
<td>Too many probabilities.</td>
</tr>
<tr>
<td>'BW'</td>
<td>Too many increments.</td>
</tr>
<tr>
<td>'BX'</td>
<td>No number to generate!</td>
</tr>
<tr>
<td>'BY'</td>
<td>Blank value line.</td>
</tr>
<tr>
<td>'BZ'</td>
<td>Genloop with no arguments!</td>
</tr>
<tr>
<td>'CA'</td>
<td>'Type' argument value invalid.</td>
</tr>
<tr>
<td>'CB'</td>
<td>'Init' argument value invalid.</td>
</tr>
<tr>
<td>'CC'</td>
<td>'Initall' argument value invalid.</td>
</tr>
<tr>
<td>'CD'</td>
<td>'Endtouch' argument value invalid.</td>
</tr>
<tr>
<td>'CE'</td>
<td>'Report' argument value invalid.</td>
</tr>
<tr>
<td>'CF'</td>
<td>'Noreplacement' argument value invalid.</td>
</tr>
<tr>
<td>'CG'</td>
<td>'Report' argument value invalid.</td>
</tr>
<tr>
<td>'CH'</td>
<td>'Singleline' argument value invalid.</td>
</tr>
<tr>
<td>'CI'</td>
<td>Blank stored line.</td>
</tr>
<tr>
<td>'CJ'</td>
<td>First non-whitespace line is not 'begin genesis'.</td>
</tr>
<tr>
<td>'CK'</td>
<td>A line is outside the global/program/features section.</td>
</tr>
<tr>
<td>'CL'</td>
<td>More than one global.</td>
</tr>
<tr>
<td>'CM'</td>
<td>Varlist was not initiated yet.</td>
</tr>
<tr>
<td>'CN'</td>
<td>Invalid stored line.</td>
</tr>
<tr>
<td>'CO'</td>
<td>feature has recursion. A calling A</td>
</tr>
<tr>
<td>'CP'</td>
<td>feature has circular recursion. A calling B calling A</td>
</tr>
<tr>
<td>'CQ'</td>
<td>Varlist was not initiated yet for add.</td>
</tr>
<tr>
<td>'CR'</td>
<td>Varlist was not initiated yet for remove.</td>
</tr>
<tr>
<td>'CS'</td>
<td>Varlist was not initiated yet for add all.</td>
</tr>
<tr>
<td>'CT'</td>
<td>Varlist was not initiated yet for remove all.</td>
</tr>
<tr>
<td>'CV'</td>
<td>Varlist needs an argument value.</td>
</tr>
<tr>
<td>'CX'</td>
<td>Genif with no arguments!</td>
</tr>
<tr>
<td>'CY'</td>
<td>Genelsif with no arguments!</td>
</tr>
<tr>
<td>'CZ'</td>
<td>Feature cannot be named global.</td>
</tr>
<tr>
<td>'DA'</td>
<td>Inconsistency. Either give all probabilities this distribution, or not at all.</td>
</tr>
<tr>
<td>'DB'</td>
<td>Inconsistency. Either give all probabilities this distribution, or not at all.</td>
</tr>
<tr>
<td>'DC'</td>
<td>Probabilities not between 99 and 101.</td>
</tr>
<tr>
<td>'DD'</td>
<td>Reference found, no feature has this Genesis name.</td>
</tr>
<tr>
<td>'DE'</td>
<td>String has characters. Put it around quotes or fix the reference.</td>
</tr>
<tr>
<td>'DF'</td>
<td>Probably an endless loop of replacement.</td>
</tr>
<tr>
<td>'DG'</td>
<td>Varlist source does not exist.</td>
</tr>
<tr>
<td>'DH1/DH2'</td>
<td>Reference found with a non-existent Genesis name.</td>
</tr>
<tr>
<td>'DI'</td>
<td>Not a proper argument for a varlist</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Code #</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DJ'</td>
<td>Value exists but has no sampled value yet.</td>
</tr>
<tr>
<td>'DK/DK1'</td>
<td>Value exists but has no sampled value yet.</td>
</tr>
<tr>
<td>'DL'</td>
<td>Either infinite recursion or an error with the compiler.</td>
</tr>
<tr>
<td>'DM'</td>
<td>The LHS value does not exist.</td>
</tr>
<tr>
<td>'DN'</td>
<td>Distribution in genmath statement does not exist.</td>
</tr>
<tr>
<td>'DP'</td>
<td>No number used for array size declaration.</td>
</tr>
<tr>
<td>'DQ'</td>
<td>No number used for array value reference.</td>
</tr>
<tr>
<td>'DR'</td>
<td>Number used for non-array value.</td>
</tr>
<tr>
<td>'DS/DS2'</td>
<td>Array number out of bounds: over.</td>
</tr>
<tr>
<td>'DT'</td>
<td>Array number out of bounds: under (-1).</td>
</tr>
<tr>
<td>'DU'</td>
<td>No index brackets used for array value.</td>
</tr>
<tr>
<td>'DV'</td>
<td>Varlist’s Value arg not a number.</td>
</tr>
<tr>
<td>'DW'</td>
<td>Number for Varlist’s value arg is out of bounds: over.</td>
</tr>
<tr>
<td>'DX'</td>
<td>Number for Varlist’s value arg is out of bounds: under.</td>
</tr>
<tr>
<td>'DY'</td>
<td>Varlist’s value arg needs an index.</td>
</tr>
<tr>
<td>'DZ'</td>
<td>Real argument invalid.</td>
</tr>
</tbody>
</table>

Table A.2: Table of Errors

The Genesis Perl preprocessor also contain bad flags/warnings, as listed in Table A.3. Some warnings, such as Warning A, lead to the removal of the instance program, while others are allowed and purely informational to the user.
### Table A.3: Table of Warnings

<table>
<thead>
<tr>
<th>Code #</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>'WB'</td>
<td>&quot;Value&quot;, Sampled at end of feature.</td>
</tr>
<tr>
<td>'WC'</td>
<td>&quot;Varlist&quot;, Sampled at end of feature.</td>
</tr>
<tr>
<td>'WD'</td>
<td>&quot;Variable&quot;, Sampled at end of feature.</td>
</tr>
<tr>
<td>'WE'</td>
<td>&quot;Distribution&quot;, Sampled at end of feature.</td>
</tr>
<tr>
<td>'WF'</td>
<td>Empty Distribution.</td>
</tr>
<tr>
<td>'WG'</td>
<td>Distribution declared but not used.</td>
</tr>
<tr>
<td>'WH'</td>
<td>No main segment.</td>
</tr>
<tr>
<td>'WJ'</td>
<td>No global segment.</td>
</tr>
<tr>
<td>'WK'</td>
<td>Code snippet in global section will not be used.</td>
</tr>
<tr>
<td>'WL'</td>
<td>Code snippet in program section will not be used.</td>
</tr>
<tr>
<td>'WDD'</td>
<td>Sampling from an Empty Varlist.</td>
</tr>
</tbody>
</table>
Bibliography


