Abstraction-Based Genetic Programming

By

Franck J.L. Binard

A thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements for the degree of

PhD in Computer Science

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

Faculty of Engineering

University of Ottawa

Abstract

This thesis describes a novel method for representing and automatically generating computer programs in an evolutionary computation context. Abstraction-Based Genetic Programming (ABGP) is a typed Genetic Programming representation system that uses System F, an expressive λ-calculus, to represent the computational components from which the evolved programs are assembled. ABGP is based on the manipulation of closed, independent modules expressing computations with effects that have the ability to affect the whole genotype . These modules are plugged into other modules according to precisely defined rules to form complete computer programs. The use of System F allows the straightforward representation and use of many typical computational structures and behaviors (such as iteration, recursion, lists and trees) in modular form. This is done without introducing additional external symbols in the set of predefined functions and terminals of the system. In fact, programming structures typically included in GP terminal sets, such as if_then_else, may be removed and represented as abstractions in ABGP for the same problems. ABGP also provides a search space partitioning system based on the structure of the genotypes, similar to the species partitioning system of living organisms and derived from the Curry-Howard isomorphism. This thesis also presents the results obtained by applying this method to a set of problems.

Acknowledgments

I would not have been able to complete this work without the encouragements, trust and support of my supervisor, Dr. Amy Felty. I want to thank Dr. Franz Oppacher, for introducing me to the field of Genetic Programming and for making me want to be a part of it. I also want to thank Vasanta Sena Lee for her careful editing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Genetic Programming (GP) [45, 46, 48] is an Evolutionary Computation (EC) search strategy in which solutions are represented as executable parse trees. GP systems evolve populations of parse trees using a selection process linked to the performance of the associated programs on a particular problem. Each parse tree in a GP population is associated with a program and the program's execution provides the metrics used to grade the quality of the tree. In this thesis, we use vocabulary from genetics and we say that the trees are *genotypes* while the programs associated with the trees are *phenotypes*.

This work is set in the context of the GP genetic representation problem and concerns the exploitation of recent advances in type theory to extend the currently used tree-based representation schemes. It describes Abstraction-Based Genetic Programming (ABGP), a search strategy related to GP in which the genotypes are compositions of computational blocks, assembled from a pattern derived from a second-order logic proof. The blocks (which are called *alleles*) are System F [28, 66] terms. System F is an extension of the simply typed λ-calculus.

1.1 Motivations and Contributions

The scientific contributions of this work are:

1. The formulation of ABGP, a GP variant that addresses a series of limitations in the classical version. In particular, the system incorporates higher-order constructions, allowing it to express data types and recursive structures without the need for extensions. This thesis presents the case for such a system and provides all the necessary theoretical details for its construction.

2. The design and implementation of an ABGP system using OCamL. The OCamL source code resulting from this research is provided in the public domain and is also a contribution of the work

3. A description of the results obtained by experimenting with ABGP.

Our primary motivation for the formulation of ABGP is the need for a GP system in which the underlying programming language is naturally capable of expressing the computational tools typically available to human programmers, in particular data types, looping structures and recursive structures. This is difficult to

achieve in GP where programs are typically represented as non-modular tree constructions and new programs are created by splitting existing programs (chosen non-deterministically) at random points and randomly recombining the program chunks obtained in this manner into new programs. This operation which is called *crossover*, resembles the method of non-sexual propagation called grafting much more than it resembles sexual reproduction. As no information about the genotype's structure is kept or used by GP, the kind of complex genotypic sub-structures required to express data types, recursion or looping are too fragile to become stable genetic material able to evolve and be reused. Abstraction-Based Genetic Programming corrects this default by providing a search space partitioning system based on the structure of the genotypes, similar to the species partitioning system of living organisms. In addition, by using System F to encode the unit computational blocks of the system, Abstraction-Based Genetic Programming provides a method to express the computational tools typically available to human programmers as closed blocks that may be plugged into other blocks. This thesis presents the results we obtained by applying this method to a set of problems.

1.2 SYSTEM F AND λ-CALCULI

System F is a λ-*calculus*. The first "untyped" λ-calculus was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s. It is a formal system designed to investigate function definition and application as well as recursion. It has since emerged as a valuable tool in computability or recursion theory. Most programming languages are rooted in the λ-calculus [52], which provides the basic mechanisms for procedural abstraction and procedure (subprogram) application. The calculus is an idealized, minimalist programming language capable of expressing any algorithm.

Typed λ-calculi can be seen as refinements of the untyped calculus which has limitations, but they can also be considered the more fundamental theory, with the untyped calculus a special case with only one type. Various typed λ-calculi have been studied. Modern functional languages, building on λ-calculi, include Erlang, Haskell, Lisp, ML, Scheme and OCaml.

System F is an extension of the simply typed λ-calculus obtained by the introduction of a universal quantification operation on types.

1.2.1 Types

Types arise naturally, even starting from untyped universes, in any domain to categorize objects according to their usage and behavior [10]. A type is a collection of values that share some properties [57]. A type system has as its major purpose to avoid embarrassing questions about representations, and to forbid situations where these questions might come up. In mathematics as in programming, types impose constraints which help to enforce correctness.

Notation for Types: In this work, we use the following conventions regarding types: the names of types begin with capital letters (as in $X, Y, Int, Bool$) and are enclosed in square brackets. f of type $[A \rightarrow B]$ is a function that takes an object of type $[A]$ as argument and returns an object of type $[B]$. f of type

$[A \rightarrow (B \rightarrow C)]$ is a function that takes an object of type $[A]$ as argument and returns an object of type $[B \rightarrow C]$. For types, the arrow is right associative, so $[A \rightarrow (B \rightarrow C)]$ is equivalent to $[A \rightarrow B \rightarrow C]$.

1.2.2 GP and Types

A problem with using GP to solve large and complex problems is the considerable size of the search space [33]. Montana [59] illustrated how the size of the search space of possible parse trees might be in the order of 10^{27} parse trees even for small problems. A type in a metaphysical sense is a category of being. For example, a mammal is a type of things. In their traditional form, GP systems do not have type structures. GP research suggests that the search space of untyped GP systems is often larger than needed because it contains both type-correct and type-incorrect programs [59, 82]. The same research experiments with typed representation in GP as a way to reduce the size of the search space. This approach was originally suggested by Koza [51] and then extended by Montana who has developed Strongly Typed Genetic Programming (STGP) [59]. There are several inter-related meanings of types. In this work, we use all of them at different times and in different contexts, so we will clarify immediately the nuances.

1.2.3 Type Theory

Intuitionistic type theory [55] is a logical system and a set theory based on the principles of mathematical constructivism. Introduced by Per Martin-Löf in 1972, intuitionistic type theory is based on the analogy between propositions and types: a proposition is identified with the type of its proofs. This identification is usually called the Curry-Howard isomorphism [23]. The Curry-Howard isomorphism was originally formulated for propositional logic and the simply typed λ-calculus. The types of type theory play a similar role as sets in set theory but functions definable in type theory are always computable. The most obvious application of type theory is in constructing type checking algorithms in the semantic analysis phase of compilers for programming languages. Type theory is also widely in use in theories of semantics of natural language. Intuitionistic type theory developed the notion of dependent types and directly influenced the development of the calculus of constructions [15] and the logical framework LF [62]. A number of popular computer-based proof systems are based on type theory, for example NuPRL [1], LEGO [64] and Coq [5]. The work presented in this thesis extends the applicability of type theory to GP.

1.2.4 Abstract Data Types (ADT)

An *Abstract Data Type* is a description of a common representation of data. It is a building block for data structures. Abstract data types are one of the most important concepts in all programming, because they allow building representations for complex data from simpler parts. An ADT is called abstract because it can be completely specified without dealing with details of implementation. An ADT leaves some aspects of its own structure undefined, to be provided by the user of the data type. System F can express ADT (see section 5.4).

1.3 ABSTRACTION

Abstraction is the generalization obtained by reducing the information content of a concept in order to retain only information which is relevant for a particular purpose. For example, a red bicycle may be studied as a

bicycle (removing the color information component), or as a vehicle (removing all information related to its particularities) or even more abstractly, as a human-made artifact. In humans, the capacity to abstract and to express abstractions is strongly related to learning and intelligence as it is the main tool that allows the inference of generalizations from specific occurrences. We chose System F as our representation scheme for several reasons (listed in Chapter 4), but mainly because of its abstraction power. There are several levels of abstractions, which we describe in this section.

1.3.1 Functional Abstraction

Functional abstraction is the particular brand of abstraction that allows a programmer to write a function that will perform a computation generically in terms of one or more named parameters. Functional abstraction facilitates reuse as it allows a programmer to avoid having to repeatedly write the same calculation. Once the function has been written, it can be instantiated as needed, by providing values for the parameters in each case. Functional abstraction is a key feature of essentially all programming languages [63]. System F uses the symbol λ to denote anonymous function abstraction. For example, given a function f of type $[A \rightarrow A]$ (a function that takes an object of type $[A]$ as argument and outputs another object of type $[A]$), we would write the System F program:

$$(\lambda x^{A}.f(f\ x)) \tag{1.1}$$

as a functional abstraction of the computation that applies f twice to an argument.

1.3.1.1 Higher-order Functions

Higher-order functions are functions which do at least one of the following:

- take one or more functions as an input
- output a function

The derivative in calculus is a common example, since it maps a function to another function. Languages that truly support higher-functions do not need to differentiate between lower-order and higher-order functions. For example, a function which takes two numbers as arguments and outputs a number has its type expressed as $[Number \rightarrow Number \rightarrow Number]$ which can be read as being the type of a higher-order function that takes a single number as input and returns a function as its output. The output function is a function that takes a single number as its only parameter and returns a number. This allows the formation of well-defined partial functions such as $(+\ 5)$, of type $[Number \rightarrow Number]$. There is a small body of GP related work [81, 84] that suggests that program representation which supports higher-order functions and abstraction is more effective than the basic GP system when applied to some problems such as the general even parity problem. System F supports higher-order functions, so the program 1.1 may be further abstracted by writing:

$$(\lambda y^{A \rightarrow A}.\lambda x^{A}.y(y\ x)) \tag{1.2}$$

Program 1.2 expresses the computation of the double application of any function of type $[A \rightarrow A]$ to an object of type $[A]$.

1.3.2 Polymorphism and Type Abstraction

System F supports an even stronger form of abstraction. The program 1.2 is only applicable in the context of some existing type $[A]$. By adding another abstraction operator, Λ, System F is able to express an even more general abstraction of the computation of program 1.2:

$$(\Lambda X.\lambda f^{X\to X}.\lambda x^{X}.y(y\ x)) \tag{1.3}$$

Program 1.3 expresses its computation independently of type. This is a *polymorphic* function. This form of abstraction, which is called *type abstraction*, allows, for example, the definition of operations on lists of objects of any type. Type abstraction is called polymorphism in the functional programming world and "generic programming" in the imperative programming world (where confusingly polymorphism means something else). There is a particularly strong kind of polymorphism, called *parametric polymorphism* [73]. Parametric polymorphism allows the definition of functions which have uniform behavior for all types. For example, a function f, defined in English as: "a function that takes two arguments of the same type and returns the first of these arguments" is parametric-polymorphic and its behavior is defined on all possible arguments, including other polymorphic functions such as itself. In System F, the f function would have type $[\Pi X.X \to X \to X]$ which is the type of the functions that take two arguments of some type $[X]$ and return an object of type $[X]$ as their output. $(\Lambda X.\lambda x^{X}.\lambda y^{X}.x)$ is an example of a term of this type and is the direct System F translation of the sentence "Take a type X as argument and two arguments, x and y, each of type X and return x" which is exactly the definition of the f function. Instantiating a function with type variables is done via a *type application* operation (see section 5.3). For example, the instantiation of f with the type $[Int]$ yields the function $(\lambda x^{Int}.\lambda y^{Int}.x)$, obtained by removing the abstraction and by replacing all bound instances of the X type variable by the type Int. This new function might also be written in its non-normalized application form $(f\ [Int])$, and its type is $[Int \to Int \to Int]$.

1.4 Organization of the Thesis

This work is structured as follows: Section 2 provides background on the general GP paradigm; Section 3 summarizes more specific related previous work; Once all necessary background has been presented, Section 4 further motivates this work and Section 5 describes System F; Section 6 is a description of the Abstraction-Based Genetic Programming System, the System F based GP system that we built to support this research and Section 7 presents and describes the experiments and results obtained using Abstraction-Based Genetic Programming. Section 8 is the concluding discussion.

Chapter 2

GP General Concepts

Computer scientists have wanted to give computers the ability to learn since the 1950s. The term *Machine Learning* (ML) was coined in 1959 by Samuel to mean computers programming themselves [68]. A good contemporary definition of ML is due to Mitchell: "Machine learning is the study of computer algorithms that improve automatically through experience" [58].

Genetic Programming (GP), a subset of ML, aspires to induce a population of computer programs that improve automatically as they experience the data on which they are trained. The first results obtained using GP methodology were reported in Stephen F. Smith's PhD dissertation [71] in 1980. In 1981, Richard Forsyth [26] described the evolution of small programs with applications to forensic science. The first modern statement of GP was given by Nichael L. Cramer [16] in 1985. Independently, Jürgen Schmidhuber, an undergraduate student evolved computer programs through genetic algorithms. The method was published in 1987 as one of the first papers in the emerging field [69]. John Koza [45, 46, 48] later explored program creation by means of evolution and established the field of GP. Since then, there has been extensive demonstration of GP as a domain-independent method to solve problems. There are many different flavors of GP. In this section, some basic concepts are presented.

GP is based on a model of what is understood to be the mechanisms of organic evolutionary learning. The principle elements of that model are:

1. Innovation caused by mutation
2. Natural selection
3. Propagation of advantageous genetic material in the gene pool via sexual reproduction

Given a problem, a GP system is initialized by the pseudo-random generation of a population of parse trees (called *genotypes* expressed in a language specified by the system's designer to fit the problem domain. Each tree in the population is then compiled into an executable program. The programs are executed in a virtual environment and each program's execution provides a measure of how close it is to being an optimal solution to the problem. This measure is the *fitness* of the program.

The fitness is used to build a probability function that assigns a selection probability to each genotype in the population. For a genotype, being selected means that parts of its code will be used in the construction of one or more new genotypes, giving it a chance to spread its genetic material in the population. This non-deterministic selection process induces a cyclic sieving effect in which lower quality genotypes tend to be removed from the population, making room for new genetic combinations assembled from higher quality genotypes.

2.1 BASIC DEFINITIONS: GP SYSTEM, GENERATIONS, RUN

In this section, we provide a general mathematical description of the GP paradigm. The definitions presented are derived and adapted from various GP literature sources. We've generalized the usual definitions so that as the Abstraction-Based Genetic Programming representation scheme and recombination methods are introduced, we may highlight the parts where the other schemes intersect and the parts that are specific to Abstraction-Based Genetic Programming. In this work, we will consistently use the following terms in the following contexts:

Definition 2.1.1 (GP System) A GP system $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$ is a tuple in which Ge is a set of parse trees called *genotypes* and Ph is a set of computer programs that can be executed by the *execution function*, $\chi_{(Ph)}$. The elements of Ph are called *phenotypes* and:

1. $\chi_{(Ph)}$ is the execution function. It executes elements of Ph in an execution environment and may return values or apply side-effects. Ph is defined as the domain of the $\chi_{(Ph)}$ function.

2. υ is a compiler that makes programs out of parse trees. $\upsilon : \Gamma \rightarrow Ph$ (where $\Gamma \subseteq Ge$) is a function that maps some or all of the genotypes in Ge to computer programs. υ may map two different genotypes to the same phenotype, but for every phenotype p in Ph, there is at least one $g \in Ge$ such that $\upsilon(g) = p$.

3. The domain of υ, Γ is defined as the elements $g \in Ge$ for which the computer program $\upsilon(g)$ can be executed by $\chi_{(Ph)}$. This is the set $\{g | g \in Ge \text{ and } \upsilon(g) \in Ph\}$. If a genotype is not an element of Γ, υ might still produce an output program, but $\chi_{(Ph)}$ will not be able to execute it and it therefore will not be an element of Ph.

4. The objective function $\varphi : Ph \rightarrow \mathbb{R}^+$ is a function that takes a phenotype p as its input and returns a value corresponding to how close the behavior resulting from the execution of $\chi_{(Ph)}(p)$ resembles the expected behavior of a solution.

GP is a search method and given $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, Ge is the search space of GP_{sys}, Ph is the solution space and the system is searching for an *optimal* element of Ge, a genotype with the following properties:

Definition 2.1.2 (Optimum Genotype) Given a GP system,
$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, an *optimum genotype* is an element $g_o \in \Gamma$ such that
$\forall g \in \Gamma, \varphi(\upsilon(g_o)) \geq \varphi(\upsilon(g))$

This definition holds, even for minimization problems because our definition for the objective function is stated in terms of *distance* from the optimum solution. Such a distance can always be obtained by transformation of the original objective function, even for minimization problems. In practice, there are usually many

optimum genotypes. A generation is the subset of the search space that is being sampled at a given time by the GP system in its search for an optimum genotype. Each new generation of genotypes is constructed using the genetic material contained obtained from the genotypes from the previous generation:

Definition 2.1.3 (Generation) Given a GP system $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, a *generation* is a pair $\langle Ge^{'}, Ph^{'} \rangle$ such that υ is a total surjective function from $Ge^{'} \subset Ge$ into $Ph^{'} \subset Ph$.

The *evolution function* is the function that produces a new generation from an existing generation:

Definition 2.1.4 (Evolution Function) Given a GP system,
$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, an *evolution function* is a non-deterministic function that takes a generation as input and outputs another generation.

New generations are constructed from the genetic material present in previous generations, mostly selected using the information obtained by the objective function. A *run* is a sequence of generations:

Definition 2.1.5 (GP run) Given a GP system
$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$ and an evolution function Φ, a run is a sequence
$(\langle Ge_0, Ph_0 \rangle, \ldots \langle Ge_n, Ph_n \rangle$ of generations such that only the genotypes of $\langle Ge_n, Ph_n \rangle$ may contain one (or several) optimum genotype(s) and $\forall i < n, \langle Ge_{i+1}, Ph_{i+1} \rangle \in \Phi(\langle Ge_i, Ph_i \rangle$. If $\langle Ge_n, Ph_n \rangle$ contains an optimum genotype then the run is said to be successful.

At any time during the run, the system manages a set of candidate solutions. Each candidate solution is a program whose behavior doesn't correspond to the program the system is searching for. If and when one of the candidate solutions produces "correct behavior", the run is halted and the program is the output of what is now a successful run. While the objective value of its associated phenotype is closely related to the amount of genetic material a given genotype will spread to the next generations, it is not the only measure. If it was, GP systems would run into problems of diversity and stagnation where some very high quality genotypes would prevent much lower quality genotypes to evolve into new original solutions. There may also be a desire to guide the evolutionary path of the search with some scaling and transformation operations such as parsimony measures, and this is done from outside the objective function. The quantity of genetic material that will be transmitted to future generations by a genotype is probabilistically determined by its *fitness*. Fitness refers to an individuals ability to compete within an environment for available resources. Goldberg describes the fitness function as some measure of profit, utility, or goodness that we want to maximise [31]. The fitness of a genotype is calculated by the evaluation function:

Definition 2.1.6 (Evaluation Function) Given a GP system
$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, the evaluation function ρ of GP_{sys} is a function that takes a genotype $g \in \Gamma$ as input and outputs a value called the *fitness* of g.

The objective and fitness functions are related values in that the fitness value of a genotype takes into account the objective value of its associated phenotype but may also take other factors into account, such as "time to failure" [65] or "time to balance" [48]. To avoid verbose functions as solutions, parsimony can be included as a fitness criterion. This technique can be used to reduce the complexity of the genotypes being evaluated [50]. In fact, in the application of genetic programming to feature discrimination, it was observed that a high degree of correlation exists between tree size and performance and that among the set of pretty good solution trees, those with the highest performance were usually the smallest within that set. It was also observed that most runs achieved a point where the size and complexity of trees eventually began to grow increasingly

larger, while performance tapered off to lower values [65, 86, 74].
The fitness value of a genotype in a current generation is a synthesis of the information needed by the evolution function to decide how much of the genotype's genetic material should be present in the next generation.

2.1.1 Two Different Types of GP Systems

In addition, we distinguish between the following two kind of GP systems:

1. *value GP systems*, in which the phenotypes compute a value (or a vector of values). For example symbolic regressors and image analyzers that try to detect objects in pictures are value GP systems. In these type of systems, the objective function is a measure of how close the value computed by a phenotype is to a known solution value.

2. *side-effects GP systems* in which the phenotype's execution produces side effects in a (possibly dynamic) environment. Taken as a whole, these side effects are the *behavior* of the program associated with the genotype, and it is the behavior that is evaluated for fitness. Robot soccer playing programs, ant simulations and motion evolution are examples of side-effect GP systems. The system may be built in such a way that the evaluation of a phenotype affects the subsequent evaluations of the other phenotypes in the population (for example in cooperation evolution or predator-prey models).

2.2 Representation and Search Space

The representation scheme and the search space of a GP system are linked in that the search space of a GP system is completely specified by its representation scheme:

Definition 2.2.1 (Representation Scheme) Given a GP system,
$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, a *representation scheme* is a set of syntactic objects and combination rules that completely specify Ge.

2.2.1 The Classical GP Representation Scheme

There are several representation schemes for GP. In this section, we introduce the first [45] and most common, which we refer to as the "classical representation scheme" in the remainder of the thesis. There are two characteristics of the classical representation scheme that we ask the reader to keep in mind as this document progresses toward the motivation section (chapter 4):

1. It does not distinguish genotypes from phenotypes, i.e. the search space is regarded as being the same as the solution space ($Ge = Ph$). Parse trees represent both the genotype and phenotype of individuals as they are modified by the genetic operators and are evaluated by the fitness function [80].

2. It does not provide any obvious, mathematically natural way to partition the search space [53].

The scheme constructs tree-like genotypes such as the one of figure 2.1 by assembling "functions" (inner nodes) and "terminals" (leaves):

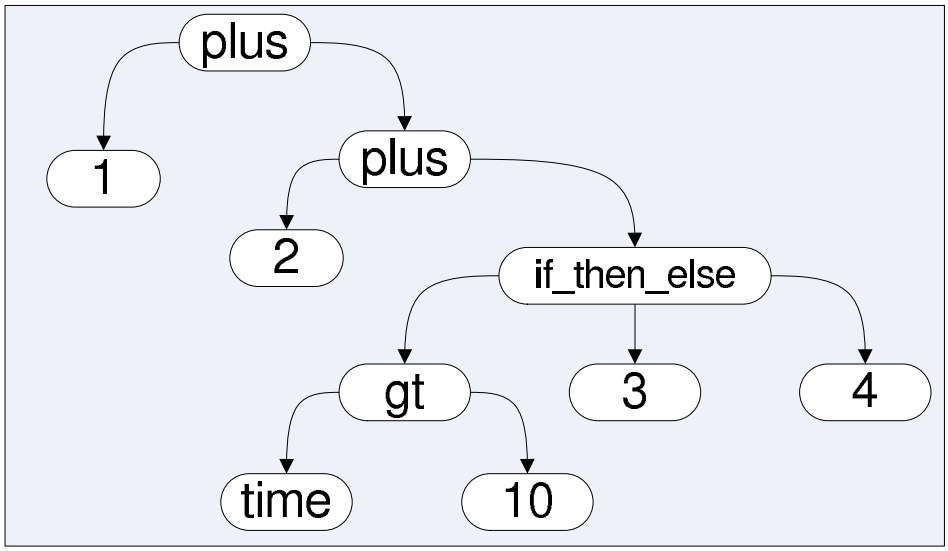

Figure 2.1: Genotype formed using the classical GP representation scheme and built from primitives defined in table 2.1

Functions	**Arity**
if_then_else	3
$plus$	2
$minus$	2
div	2
gt	2
Primitives	**Arity**
$[0, \ldots, 10]$	0
$time$	0

Table 2.1: A primitive set $\Im$ that can be used to represent the genotype of figure 2.1

Definition 2.2.2 (Primitive Set Definition (Classical)) A set $\Im$ of symbols for a GP system, is a set of elements (f, n) such that f is a symbol, $n \geq 0$ is an integer, and there is at least one element $(f^o, 0)$ in $\Im$. Given (f, n), we call f a *function* when $n > 0$ and a *terminal* when $n = 0$.

In $(f, n) \in \Im$, n is the *arity* of the symbol f. It is necessary to include a symbol with arity 0 in F because genotypes are trees of nodes labeled by symbols existing in F, and the node branching is exactly the arity of the symbol. Because classical GP is untyped, the arity of a symbol is the only specification that indicates how the symbol may be used in a genotype. When the arity is 0, the symbol is used as a leaf and leaves are constants. When its arity is greater than 0, the primitive is a function and is used as an inner-node. We are now ready for the definition of a genotype in this representation scheme:

Definition 2.2.3 (Genotype Definition (Classical)) Given a set of primitives F, a *genotype* is a tree with root node labeled f (where $(f, n) \in F$) and n children nodes. Each child node is itself a genotype (which we'll call a *sub-genotype* here).

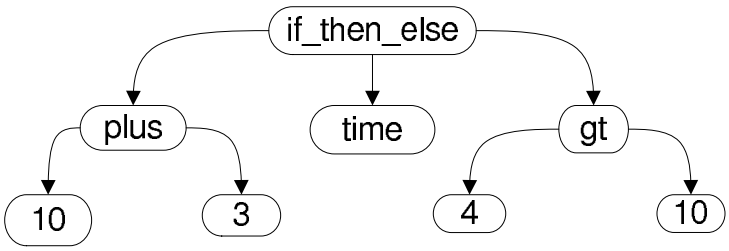

Figure 2.2: This genotype is formed legally using the set of primitives of table 2.1. It is non-sensical when used with the $\chi_{(Ph)_1}$ function defined in table 2.2, but is sensical when the $\chi_{(Ph)_2}$ function is used.

Using definition 2.2.3, the genotype represented in figure 2.1 can be expressed if and only if the symbols $(if\_then\_else, 3)$, $(plus, 2)$,$(gt, 2)$,$(time, 0)$,$(1, 0)$,$(2, 0)$,$(3, 0)$,$(4, 0)$,$(10, 0)$ are included in F. Table 2.1 lists the elements of a set of primitives able to express the genotype of figure 2.1.

2.2.2 Search Space Considerations

2.2.2.1 Program Complexity

In a system $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, Ge is typically infinite. Often, in GP, the system might consider only a finite subset of Ge as its search space. This is usually done by defining a *complexity* measure on elements of Ge and considering only the genotypes that have a complexity value under some pre-defined threshold. There are various measures of complexity, but usually complexity is related to the size of the genotype, for example the number of nodes it contains or the depth of its deepest branch.

2.2.2.2 Sensicality

We've already said that given a system $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, a genotype may not be an element of Γ, yet υ might still produce an output program for it. The problem is that $\chi_{(Ph)}$ would not be able to execute it. We now consider the case of a syntactically correct parse tree for which the evaluator is incapable of producing a program that can be executed by the system's $\chi_{(Ph)}$ function. We call such a genotype *non-sensical*. We formalize the notion of *sensicality*:

Definition 2.2.4 (Sensical Genotype) Given a GP system,
$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, a genotype $g \in Ge$ is *sensical* when it belongs to Γ and non-sensical when it doesn't.

A genotype is sensical when the GP system is able to use it as input to make a program that can be evaluated to yield a fitness value. Sensicality is a problem in GP. For example, in the classical representation scheme, there are usually many more non-sensical genotypes in the search space than sensical ones [59], yet solutions can only be programs generated from sensical genotypes. There are two strategies for handling non-sensical genotypes. Given $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, we may:

1. Have a simple partitioning system that let's us know if a genotype belongs to Γ so that we may discard it if it doesn't.

Functions	$\chi_{(Ph)_1}$	$\chi_{(Ph)_2}$
(if_then_else)	Evaluates to 2^{nd} argument when evaluation of 1^{st} argument returns $< true >$ and to the 3^{rd} argument when the evaluation of the 1^{st} argument returns $< false >$	Evaluates to 2^{nd} argument when evaluation of 1^{st} argument is greater than $< 0 >$ and to 3^{rd} argument otherwise
$plus$	Evaluates to the sum of the evaluation of its 2 arguments	Evaluates to the sum of the evaluation of its 2 arguments
$minus$	Evaluates to the substraction of the evaluation of its 2^{st} argument from the 1^{st}	Evaluates to the substraction of the evaluation of its 2^{st} argument from the 1^{st}
div	Evaluates to the result of dividing the result of the evaluation of the 1^{st} argument by the evaluation of the 2^{nd} when the 2^{nd} is different than $< 0 >$ to $< 0 >$ otherwise	If the result of the evaluation of the 2^{nd} argument is different than $< 0 >$, evaluates to the result of dividing the result of the evaluation of the 1^{st} argument by the evaluation of the 2^{nd}. Evaluates to $< 0 >$ otherwise
$times$	Evaluates to the product of the evaluation of its 2 arguments	Evaluates to the product of the evaluation of its 2 arguments
gt	Evaluates to $< true >$ when the evaluation of the 1^{st} argument is greater than the evaluation of the 2^{nd} argument and to $< false >$ otherwise	Evaluates to $< 1 >$ when the evaluation of the 1^{st} argument is greater than the evaluation of the 2^{nd} and to $< 0 >$ otherwise
Terminals		
$[0, \ldots, 10]$	Evaluates to the corresponding number	Evaluates to the corresponding number
$(time)$	Evaluates to $< time >$	Evaluates to $< time >$

Table 2.2: $\chi_{(Ph)_1}$ doesn't accept the genotype of figure 2.2 but preserves the intended meaning of the functions and terminals, $\chi_{(Ph)_2}$ accepts the genotype of figure 2.2 but doesn't preserve the intended meaning of the operators.

2. Design the system in such a way that $Ge = \Gamma$, eliminating the possibility of forming non-sensical genotypes.

The problem with the first solution is that the system's pseudo-random mechanism still ends up doing all the work associated with assembling non-sensical genotypes before it checks if they are executable. Often, the only simple mechanism is to run the compiled code obtained from the genotype through $\chi_{(Ph)}$ and watch it choke. As a result, GP practitioners have trended toward the second approach. But this introduces its own sets of problems. Consider the genotype represented in figure 2.2, which uses the symbols defined in table 2.1. It is a legal but non-sensical genotype when the $\chi_{(Ph)_1}$ function of table 2.2 is used. The issue here is that the $\chi_{(Ph)_1}$ execution function generates reasonable interpretations of what a sensical program would intend to compute. Compare with the $\chi_{(Ph)_2}$ function, which avoids all possibility of forming non-sensical genotypes but corrupts the intended meaning of the operations defined in the set of primitives. With the $\chi_{(Ph)_2}$ function, the genotype of figure 2.2 is sensical, but the question may be asked: should it be? $\chi_{(Ph)_1}$ preserves the intended meaning of the elements included in the set of primitives (except for the protected div function). It does this by implicitly assuming the existence of two different type of values (booleans and numbers). Non-sensicality happens when objects and functions are not used properly with respect to their typing constraints. We may forbid the construction of non-sensical genotypes by avoiding the introduction of types (as the $\chi_{(Ph)_2}$ function does), or we may do it by introducing an explicit type system. However we will show, when we look at type GP representations, that it too may run into problems of sensicality when attempts are made to increase its expressive power by adding generic language constructs. In any event, sensicality is one of the most salient issues of the classical representation scheme.

2.2.2.3 Search Space Partitioning

Since John Holland's work in the 1970s and his schema theorem [37], *schemata* are often used to explain why genetic algorithms (GA) work. Schemata are "similarity templates" and the schema theorem describes how they are expected to propagate generation after generation. Schema theorems lead to relatively concise descriptions of the way GAs search which can be easily understood [53]. A natural way of creating

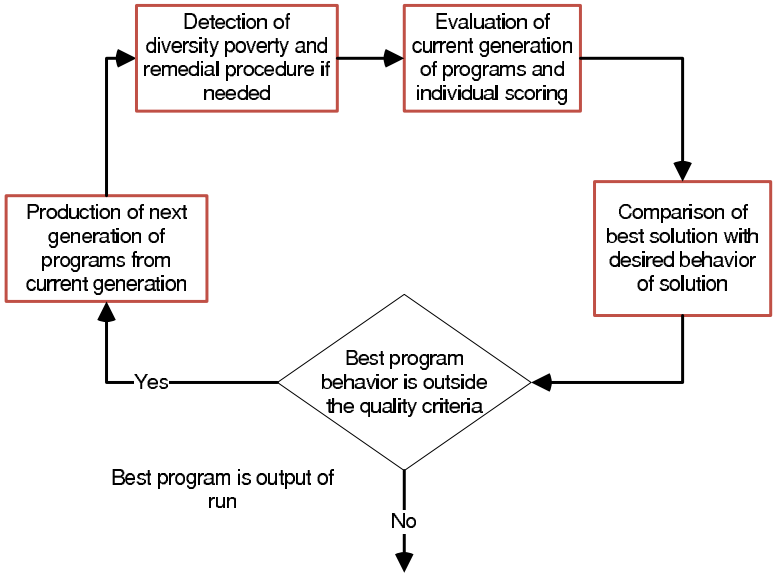

Figure 2.3: *GP system cycle*

a theory for GP is to define a concept of schema for trees and to extend Holland's schema theorem to GP. One of the difficulties of doing this is that the definition of a schema for GP is much less straight-forward than for GAs [53] and several alternative definitions of a proper way to partition a GP system's search space have been proposed in the literature. The next chapter includes a detailed survey of these efforts at partitioning the GP's search space and of the theoretical observations that can be deduced from each of these partitioning methods. For now, we will simply say that search space partitioning is used to identify subsets of the search space as a theoretical tool to study the effect of recombination on GP and that given a system $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, there are two ways to partition Ge:

1. By component inclusion: defining a sub-tree and partitioning Ge into the set of genotypes that contain the sub-tree and the set that doesn't.

2. By structure: defining a genotype meta-structure (or template) and partitioning Ge into the set of genotypes that are built on that structure and the set of genotypes that are not.

The research presented in this thesis offers a partitioning scheme that uses both methods.

2.3 From Generation To Generation

In this section, the specific details of the mechanisms by which a GP system produces new generations are provided.
A GP system, $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$ breeds computer programs to solve problems by:

1. Randomly generate an initial generation $\langle Ge_0, Ph_0 \rangle$.

2. Iteratively perform the following sub-steps until the termination criterion has been satisfied:

   (i) From the current generation $\langle Ge_i, Ph_i \rangle$, compute $\chi_{(Ph)}(p)$ for all the elements $p \in Ph_i$ and use the information derived from this computation to compute $\varphi(p)$

   (ii) using the values obtained in (a), and other considerations specific to the problem and the scheme being used, derive a fitness value $\rho(g)$ for all the elements $g \in Ge_i$

   (iii) Use the information obtained in (b) to identify the new promising subsets of Ge from which the elements of $\langle Ge_{i+1}, Ph_{i+1} \rangle$ will be chosen.

The iteration continues until an optimum genotype is found. Figure 2.3 illustrates the iterative part of the algorithm.

2.4 EVALUATION

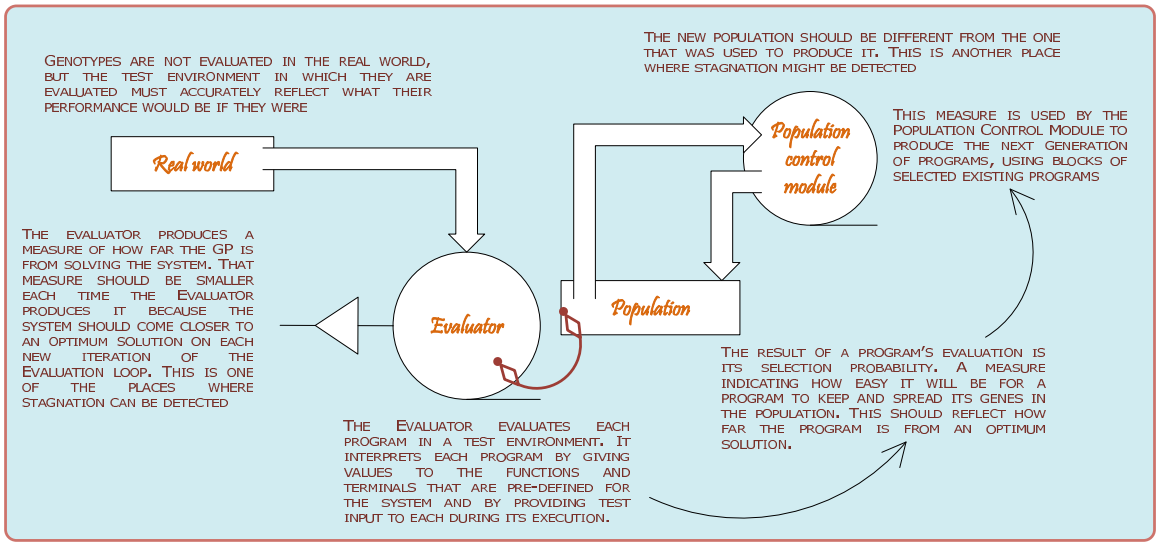

Figure 2.4: GP evaluator feed-back loop

Figure 2.4 describes the evaluation cycle of a GP system. The fitness score of a genotype takes into consideration the objective score of its phenotype. Given a system:

$$GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle \tag{2.1}$$

An evaluation function ρ assigns a *fitness* score to each genotype in a generation $\langle Ge_o, Ph_o \rangle$ of 2.1. The evaluation value of a genotype takes into consideration information as to how well its corresponding phenotype is adapted is to its environment, but it may also take into account information as to how compact the genotype is, how long it takes to execute its associated phenotype, or how it compares to the other genotypes in the generation. The fitness value is used to select the genetic material of the current generation that will be propagated into the next generation.

Operations such as scaling and normalizing are often used as components of the fitness determination operation to minimize the probability that above average genotypes will flood the population with their genes, leading to loss of diversity and ultimately to premature convergence. Scaling converts the raw objective scores to values in a range that is suitable for the selection function. The two most common types of scaling functions are:

- Rank scaling, which scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores. The rank of the most fit individual is 1, the next most fit is 2, and so on. Rank fitness scaling removes the effect of the spread of the raw scores.
- Proportional scaling makes the fitness value of an individual proportional to its objective score.

2.5 SELECTION

A GP's evolution strategy constructs the system's next generation by recombining and modifying what it considers to be promising genetic material in the current generation. The selection function identifies promising genetic material and determines how much of it should be present in the next generation. The genetic material that composes genotypes of higher quality tend to have a higher probability of finding itself reused in some form or other in the next generation. In most GP systems, selection is applied at the organism level and the term "mating pool" is taken to mean the storage containing the individuals selected for reproduction. While GP systems typically select genotypes, there are other possibilities. In particular, selecting whole sets of genotypes (such as schemata) or combinations of sets and individual genotypes is also possible. Indeed, this is the approach that our research has taken. For now, we'll present the most common selection schemes in the literature.

A *genotype selection scheme* determines the probability that a genotype will be selected for producing offspring by recombination or mutation. In order to search for increasingly fitter phenotypes, higher selection probabilities are assigned to the genotypes of better scoring phenotypes. The selection operator selects genotypes based on the general principle that the fitter the individual, the higher its probability of being selected for reproduction should be. The most popular selection methods are truncation selection, fitness-proportionate selection, ranking-based selection, tournament selection and Boltzmann-selection. Selection might operate with or without replacement. With replacement, the solution that is added to the new parent population is kept in the combined population and a good solution can be chosen more than once for the new parent population. The term with replacement is used since the original genotype is available for further selection as additional genetic material is selected. The system maintains no memory of prior selection. Without replacement, the selected solution is placed in the parent population and removed from the combined population and each solution can only be selected once for the new parent population.

2.5.1 Fitness-Proportionate Selection

Fitness-proportionate selection [37] is a stochastic selection method in which the selection probability of a genotype is proportional to its fitness. Specifically, the probability that a genotype g associated with a

phenotype p in a generation $\langle Ge^{'}, Ph^{'} \rangle$ will be selected is:

$$\frac{\rho(p)}{\sum_{p' \in Ph^{'}} \rho(p')} \tag{2.2}$$

While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. A solution may be weak, but may include some component which could prove useful following the recombination process. Fitness-proportionate selection with replacement is considered to be the standard method of selection in genetic programming, primarily because it is historically accepted as a versatile technique that has been widely implemented. In addition, this selection method has also proven successful in a variety of problem domains [43, 20] and the selection algorithm itself is relatively straightforward. Fitness-proportionate selection has the short-coming of over-select: individuals with super-fitness tend to get selected too often, leaving other individuals under-sampled. Rank-based selection (2.5.3) was proposed to avoid this problem.

2.5.2 Truncation Selection

In *truncation selection* [70] individuals are sorted according to their fitness. Only the best individuals are selected as parents. The parameter for truncation selection is the *truncation threshold*. It indicates the proportion of the population to be selected as parents and takes values ranging from 50%-10%. Individuals below the truncation threshold do not produce offspring. Unlike fitness proportionate selection there are no chances for weaker solutions to survive the selection process.

2.5.3 Ranking-based Selection

Ranking-based selection [79, 32] sorts the genotypes of a generation according to the raw fitness score of their associated genotypes. The probability of a genotype being selected for reproduction depends only on its position in terms of fitness relative to the other genotypes and not on the actual fitness score. Rank-based fitness assignment might help overcome the scaling problems of the proportional fitness assignment: stagnation in the case where the selective pressure is too small or premature convergence when the selection has caused the search to narrow down too quickly.

2.5.4 Boltzman Selection

Boltzmann selection is a method inspired by the technique of simulated annealing. In Boltzman selection, selection pressure is slowly increased over time to gradually focus the search. The inspiration comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. By analogy with this physical process, each step of this selection algorithm replaces a current individual by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T called the temperature. The temperature is gradually decreased during the process, creating a dependency such that the current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minimawhich are the bane of greedier methods.

2.5.5 Tournament Selection

Tournament selection runs a "tournament" among a few individuals chosen at random from the population and selects the winner (the one with the best fitness) for crossover. Selection pressure can be easily adjusted by changing the tournament size. If the tournament size is larger, weaker individuals are less likely to be selected.

2.5.6 Elitism

Elitism is an addition to many selection methods where one or more of the highest-fitness individuals are copied, unchanged, from one generation to the next. The ratio $\frac{N}{M}$ between the elite size, N, and the population size, M, is called the *elite fraction*. Elitism is commonly used in generational GP to ensure that the best individuals discovered in a generation are not lost, and are made available for possible further improvements to new generations. As selection is probabilistic, such individuals might otherwise be lost if they're not selected to reproduce. If the selection method guarantees the conservation of some individuals, it is called *elitist*.

2.6 GENETIC OPERATORS

Genetic operators are the tools used at each stage of the genetic program to produce new programs. This section describes a variety of genetic operators that are commonly used.

2.6.1 Recombination Operator

Recombination or *crossover* is the structural operation that produces a new program from the re-assembly of elements of existing programs.

2.6.1.1 The Resemblance Property of Crossover

In nature, genetic information encoding is done in a way that admits sexual reproduction. Asexual reproduction typically results in offspring that are genetically identical to the parent. Sexual reproduction allows the creation of genetically different offspring that are still of the same general flavour (species). In any evolutionary computation scheme, the fitness of the offspring should be related to that of the parents. Otherwise, the process would degenerate into a random search across the representation space.

2.6.1.2 Classical GP Crossover

Given two genotypes, the crossover operator produces a number of offspring. The tree-structure genetic material used in GP suggests the crossover scheme used in the classical version of GP: Given two parents, a and b, a subtree from a is chosen and is inserted in b, replacing whatever was there previously. The new tree shares characteristics from both parents. Figure 2.5 illustrates the process.

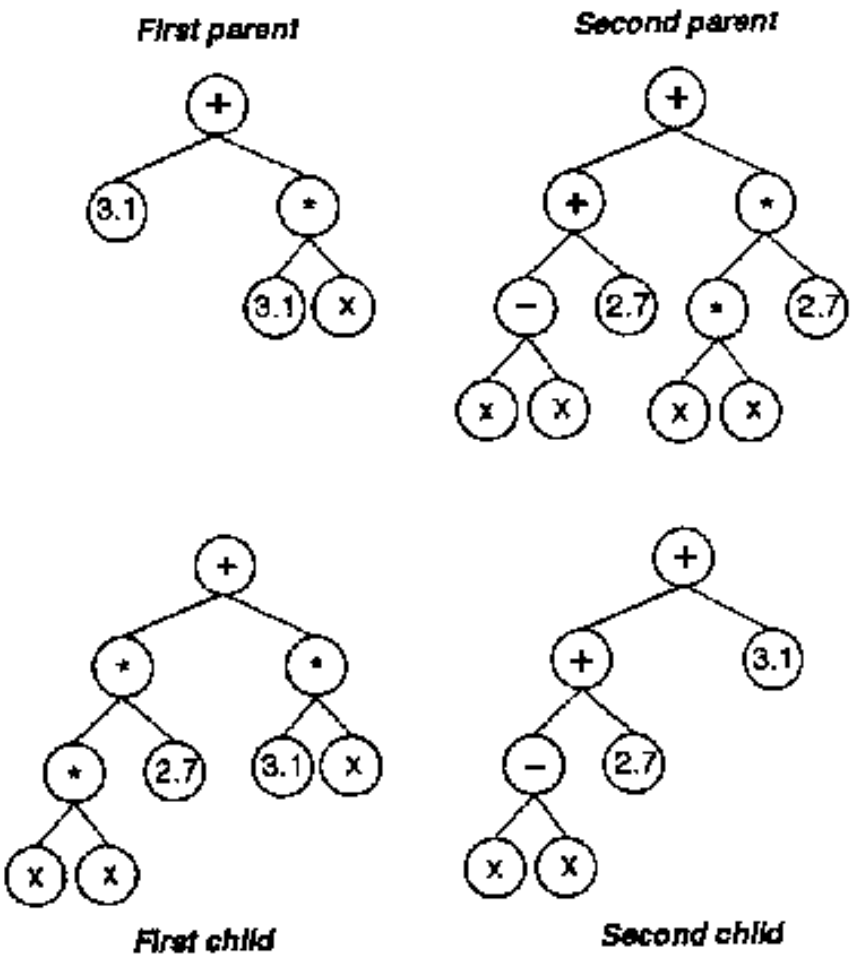

Figure 2.5: An example of crossover. A branch is selected for crossover in each of the parent trees. In this example, the first left branch from the first parent is exchanged with the first right branch of the second parent. This creates two new trees. Diagram from [19]

2.6.2 Mutation Operator

Mutation operates on only one individual. Normally, after crossover has occurred, the mutation operator is applied to the offspring with a low probability. The operator randomly selects a point position of an offspring genotype with some probability, usually small, and modifies it within a range of values.

2.7 CONCLUSION

In evolutionary computing, *evolvability* is the capability of a system to make additional progress given more computational resources until an optimum solution is reached. A run may be halted when an optimum genotype has been found, but it may be also be halted because it has been detected that the genotypes are not getting any better from one generation to the next. It is possible for the system to reach a state in which no new candidate solution appears whose fitness is better than the best program produced so far. Systems that reach this state tend to "recycle" the same genotypes over a cycle of generations. In these systems, larger

deviations in new genotypes tend to be short lived because they are unable to compete for survival with well-established but sub-optimal genotypes. The fitness function is now rewarding some easy to evolve behavior and is discouraging the evolution of some harder to evolve computation steps needed to evolve the more optimum computation. This leads to *stagnation*. This can often be detected early by a quick and chronic lack of diversity in the population. Evolvability is related to *population diversity* [25]. One of the major obstacles to achieving sustainable evolution is lack of diversity: some essential building blocks are missing in the population. Diversity is necessary for the existence of sufficient genotypic materials for assembling and mixing in a population, which is in turn necessary for sustainable evolution. This is a central concept of both natural and artificial evolution. Diversity can be produced by some reverse elitist strategy, where low-performing solutions are protected from immediate destruction. Subsidizing the existence of lower quality programs is worthwhile if it can be shown that they have characteristics worth preserving (rarety for example). Evaluation is usually the computing resource bottleneck in a GP system. As in nature where it relates to

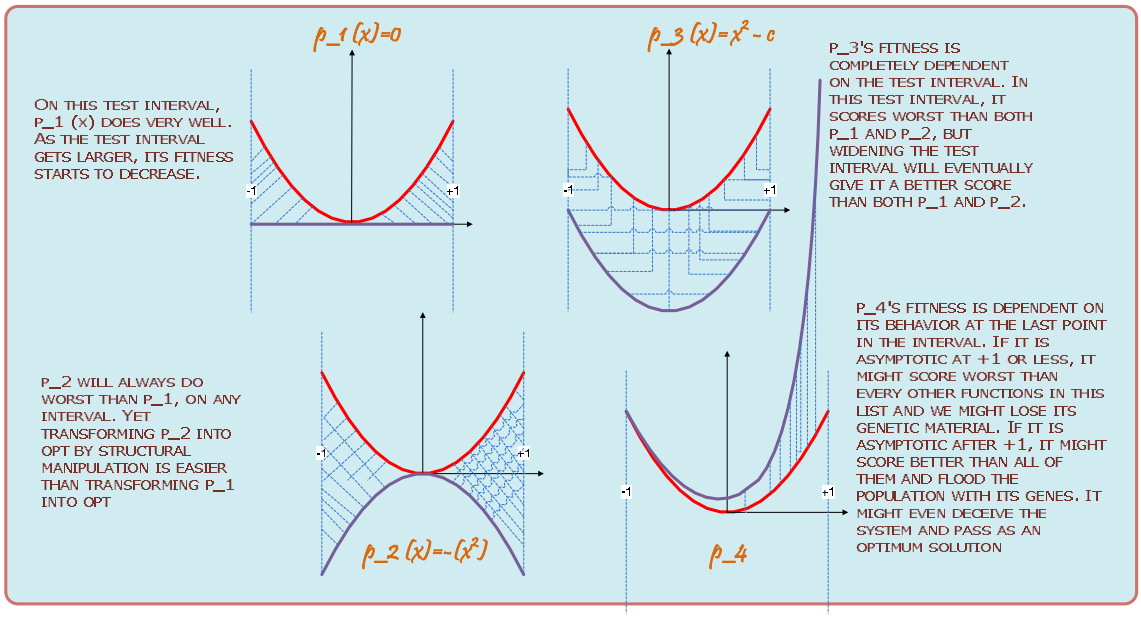

Figure 2.6: Measuring the quality of an approximation

ecological wealth, diversity is linked to the amount of resources that each program uses relative to the amount of computing resources that are available for all programs. In an environment where only one program can be executed, there can be no diversity, and selective pressure is too harsh for meaningful solutions to emerge by Darwinistic processes. In an environment where an infinity of programs can be executed, there can be perfect diversity, and the system already contains many optimum solutions in its population at generation time. We must do with something in between, but the capacity to manage and evaluate large populations is definitely an advantage in GP. Another problem with GP evaluation is that while ρ is used as a *measure* of the distance from a genotype to the optimum genotype in the search's landscape it is really a distance from a phenotype's behavior to the solution program's behavior. This raises three issues:

1. We don't have the solution program (if we did, there would be no point to the whole thing). The candidate solutions can only be scored on the finite amount of input cases for which the output is

known. This introduces the danger of computing the wrong thing because the system might find a program that fits perfectly inside the range of test cases but is a poor match outside of it. This is a common problem in machine learning and is usually refered to as *extrapolation*.

2. In figure 2.6, a symbolic regression GP system is scoring approximations to the function $opt(x) = x^2$. The problem of symbolic regression can be stated as "given a set of data points, find the underlying function in symbolic terms" [48]. More specifically, suppose the system is given a set $L = \{(x, y)|y = x^2, x \in [a, b]\}$ as data points. Typically, for this task the fitness of a program is a variation of 1 divided by the sum of the differences between the actual values of $y \in L$ and the values predicted by the system for the $|L|$ values of x. The fitness of an approximate solution $p_i(x)$ is basically a number that is inversely proportional to the area of the patterned section of figure 2.6. In this case, the test interval is $[-1, 1]$. p_4 is a function with a vertical asymptote somewhere around $x = 1$. It illustrates an approximation that looks a lot like *opt* on this particular test range, but is very different as soon as we step outside the range.

3. The only tools available to the system to construct new and hopefully better programs are structural changes to existing programs. When we look at things from the point of view of how many structural operations it takes to transform a candidate solution into opt, the measure we are using makes little sense. For example, with this metric, the program $p_1(x) = 0$ will always score better than $p_2(x) = -(x^2)$ on any range of test cases. Yet it is easier to transform p_2 into *opt* using a sequence of structural modifications. It is also much easier to express a relationship between the two: both *opt* and p_2 belong to the family of functions $\{g(x)|g(x) = a * x^2, a \in \mathbb{R}\}$. There is a similar situation with p_3. So the danger here is to over reward the p_1 function and to "lose" the p_2 and p_3 genetic material while the population gets filled by variations of the p_1 function and never explores sufficiently around the other functions.

In the next chapter, we will present the work specific to the representation problem in GP and upon which the research presented in this thesis is based upon.

Chapter 3

GP Representation Specific Work

In this chapter, more advanced specific GP concepts related to our work are presented. Of particular relevance to this thesis is the work concerning search space partitioning and the work related to typed representation.

3.1 Fitness Landscape Analysis

Given a GP system $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, each element of Ge represents a search point in the space of potential solutions to a given problem. If genotypes could be visualized in two dimensions, then a *fitness landscape* would be a three-dimensional map with the fitness of the genotypes as the height. Evolution causes populations to move along a fitness landscape in particular ways. The evolvability [2] of an evolutionary system can be seen as movement toward *local peaks* in the fitness landscape. A local peak is not necessarily the highest point in the fitness landscape, but any small movement away from it leads to decreasing fitness.

Definition 3.1.1 (Evolvability (GP context)) *Evolvability* is the ability of a population to produce variants fitter than any yet existing.

By contrast, an *optimum* is a peak from which any movement will lead to a decrease in fitness. Many evolutionary algorithms are inherently convergent and stagnate after a certain number of generations. Evolutionary algorithm practitioners handle premature convergence using methods specifically designed to slow down this degradation of search capability. These include increasing the population size, tuning crossover and mutation rates, and dynamically adapting running parameters. Unfortunately, these tricks often seem to only delay the stagnation process. [29] explains premature convergence from the evolvability point of view: When the fitness of individuals gets higher and higher, the variation of phenotypes becomes more and more costly. Selection then tends to favor individuals that increasingly conserve their fitness and exhibit less and less phenotypic variation.

An alternative approach to understanding how GP searches is based on the idea of dividing the search space into subspaces (called *schemata* [38, 30]). The original idea for schemata originates from Genetic Algorithms (GA), a branch of evolutionary computation that concerns itself with the evolution of solutions to problems (in contrast with GP that concerns itself with evolving problem solving programs). Schemata were brought

up with the objective of providing a formal model for the effectiveness of the GA search process. The concept was later adapted to GP. A schema groups together the genotypes that have certain macroscopic properties in common. Using such subspaces it is possible to study how and why the genotypes in the population move from one schema to another. The search might be studied from the point of view of how the percentage of the population that belongs to a certain schema varies from generation to generation during a run.

3.1.1 Program Component Schema Theories

Component analysis concentrates on the propagation of sub-components of genotypes. In some definitions, [47, 60, 77, 78, 61] components of a schema are *non-rooted* in the sense that the schema can potentially match components anywhere in the tree. The first definition of a schema in the context of GP appears in [47] and it applies to the classical version of GP. [47] simply defines a schema as the subset of programs of a GP population that contain a specified tree. For example, the schema $\{(+\ 1\ 2), (*\ times\ 10)\}$ represents all the genotypes of a population that include at least one occurrence of each of the sub-trees $(+\ 1\ 2)$ and $(*\ times\ 10)$. The definition specifies only the components of a schema and not their positions or number of occurrences, so the same schema may be matched in different ways by the same program. This contrasts with GA schemata in which multiple matching is not possible.

Koza's work on schemata was formalized and refined by [61, 60] who derived a schema theorem for GP with fitness proportional selection and crossover. They define a schema as a multiset of subtrees and tree fragments. They do this by adding two items to the definition of a schema:

1. a structural dimension by including tree templates called *tree fragments* that include a "wildcard symbol" ('#') that can be matched by any legal expression.

2. a number associated with the fragment (or complete legal sub-tree as this is still possible) indicating the number of times it must be included in the program in order to consider the program as an element of the schema.

This definition of schema allows the definition of the concepts of order. Order is related to the number of defining symbols (non '#'). The less defining symbols, the lower the order. GP building blocks [60] would be low order, consistently compact GP schemas with consistently above average observed performance that are expected to be sampled at increasing or exponential rates in future generations. The average number of links needed to connect a schema's subtrees for all the instantiated programs is used to calculate the defining length of the schema, a measure equivalent to the number of crossover events that may destroy the schema. *Compactness* is the inverse of the metric. The longer the average defining length of the schema is, the higher its probability of being destroyed by crossover. From these definitions, [60] derives a schema theorem, but because their definition allows genotypes to match a schema in several different manners, the defining length of a schema cannot be fixed as a constant. The conclusion that the probability of disruption of a schema is not a constant but varies on the shape, size and composition of each genotypes that match it, leads to [60]'s suggestion that the propagation and the use of building blocks may be unattainable in GP.

3.2 Related Representation Work

3.2.1 Constrained Syntactic Structures

The classical GP specification [48] requires the definitions of the primitives of the system to satisfy the *closure* requirement. Closure is the constraint that all the elements used in the genotype must be of the same type and that no evaluation step will produce a run-time error (so that for example, division by 0 is artificially made safe). Closure is satisfied when "any possible composition of functions and terminals produces a valid executable computer program" [50]. From our model's point of view this translates as saying that given $GP_{sys} = \langle Ge, Ph, \Gamma, \upsilon, \varphi, \chi_{(Ph)} \rangle$, then $Ge = \Gamma$ holds iff GP_{sys} satisfies the closure requirement. Closure usually implies using a monomorphic programming language to code the genotypes. This in turns implies carefully defining the terminals and functions so as to not introduce multiple data types. Typically, general definitions for decision functions or predicates are avoided to remove the need for a Boolean type. Instead, combined constructs such as (3.1) are used.

$$(if\_smaller\_than\_else\ x\ y\ u\ v) \tag{3.1}$$

Where x, y, u, v are of the same type. This style of function definition introduces two issues:

1. In order to increase the expressive power of the language, the system's designer is forced to define several specialized functions instead of a few general functions that may be combined to form specialized structures. This makes the syntax pre-defined for the GP system heavier than needed and it introduces a bias into the GP's search strategy.

2. It lacks flexibility. Functions defined in this way prevent the evolution of specialized problem specific programming constructs. For example, it would be impossible for a GP system that respects the closure constraint to evolve an expression such as:

$$if\_else((smaller\_than\ x\ y)\ or\ (eq\ 0\ x)), \ldots \tag{3.2}$$

Koza has realized this shortcoming and added a mechanism called *constrained syntactic structures* [48] to relax the closure property. Constrained syntactic structures define a set of problem-specific syntactic rules specifying which terminals and functions are allowed to be the child nodes of every function in the program trees. They are a way to enforce data type constraints to only generate parse trees satisfying these constraints ("legal" parse trees). Constrained syntactic structures are used for problems requiring data typing. They define problem-specific syntactic rules that specify which terminals and functions are allowed to be the child nodes of functions in GP program trees.

3.2.2 Context-free Grammar Approach

Also noting that the requirement of closure makes many program structures difficult to express, [76] proposed the use of *context free grammars* (CFGs) to specify the structure of the system's programs. A context free grammar describes the admissible constructs of a language. Note in the following definition that *non-terminals* and *terminals* have different meanings in GP and in CFG.

Definition 3.2.1 A CFG is a 4-tuple, $\{S, N, T, P\}$ where S is the start symbol, N is a set of non-terminal symbols, T is the set of terminal symbols and P are the production rules.

In CFG-based GP, program trees are derivation trees and they are generated from the grammar. At the root of each derivation tree is the start symbol S. The trees are iteratively built up from the start symbol by re-writing each non-terminal symbol into one of its derivations. Crossover is restricted to swapping subtrees built on the same non-terminal (in the CFG sense) symbol.

For rules whose non-terminal can be rewritten recursively, an upper bound of the number of recursive rewritings of this rule is specified and used to limit the size of the program tree.

3.2.3 Typed GP Systems

There are two reasons to consider using a type system as a GP representation scheme: eliminating the closure constraint and narrowing the system's search space. In the classical representation scheme, the size of the search space is a function of the cardinality of the set of primitives, of the average arity of its elements and of the size limit of the program trees. It is necessary to include enough symbols in the set of primitives to be able to express a solution to the problem for which a solution is being evolved, but increasing its size increases the size of the search space. When functions and terminals need to be embedded in a context-specific position in the genotype to correctly express something, the probability of actually assembling a correctly formed program by some pseudo-random recombination process is very small. So, instead of adding more functions and more terminals, we may consider increasing the expressive power of the system by adding less specialized functions to the function set. The problem is in ensuring that these more general structures do not increase the probability of producing non-sensical genotypes. This realization, lead to the exploration of representation schemes in which the terminals and functions are associated with a specification indicating in which contexts they may or may not be used. Attaching a type to each element of the GP's set of primitives ensures that computational blocks may combine only with other computational blocks in a sensible manner. Types can be used to specify how a block may be plugged into another block.

3.2.3.1 Strongly Typed Genetic Programming (STGP)

In general, types are used by human programmers to rule out nonsensical computations. Types can also be used for this purpose in the context of GP. This has the positive side-effect [82] of restricting the search space to allow only programs that make sense from a run-time point of view. [59] shows how types can be used to eliminate the closure constraint of un-typed genetic programs and to restrict the search space of genetic programs. Montana named the method *Strongly Typed Genetic Programming* (STGP). In STGP, each function has a specified type for each argument and for the value it returns. For example,

$$(func, [A \rightarrow A \rightarrow B]) \tag{3.3}$$

defines a function that takes 2 arguments of type $[A]$ and returns an object of type $[B]$. The descriptions of a simply-typed system given below do not exactly conform to the system that is proposed by Montana, but they are equivalent. In particular, the version described here uses functions with only two branches where Montana includes n-branching tree nodes. The simplest version of Strongly Typed Genetic Programming [59] may be formalized as:

Definition 3.2.2 (Primitive Set Definition (STGP)) Let ty be a set of symbols. A type σ on ty is either an element of ty or a construct of the form $\alpha \rightarrow \beta$ where α and β are types on ty. A set of primitives on ty for

Type	
Boolean	User Defined Atomic Type (UDAT)
Number	User Defined Atomic Type (UDAT)
Functions	**Type**
$plus$	$[Number \rightarrow Number \rightarrow Number]$
$minus$	$[Number \rightarrow Number \rightarrow Number]$
div	$[Number \rightarrow Number \rightarrow Number]$
$times$	$[Number \rightarrow Number \rightarrow Number]$
gt	$[Number \rightarrow Number \rightarrow Boolean]$
Terminals	**Type**
$[0, \ldots, 10]$	$[Number]$
$time$	$[Number]$
$[true, false]$	$[Boolean]$

Table 3.1: Types, functions and terminals definition for typed version of the terminals and functions previously defined in table 2.1

a GP system, $\Im_{ty}$ is a set of elements (f, σ) such that f is a symbol and σ is a type on ty.

We remark that this definition allows the inclusion of higher-order primitives. These are primitives that are functions that either take functions as arguments or return functions. For example, it is possible to include a primitive of type:

$$[(Number \rightarrow Number \rightarrow Number) \rightarrow Number \rightarrow (Number \rightarrow Number)]$$

Table 3.1 displays the typed version of the set of functions, terminals and constants defined in Table 2.1, with the exception of the if_then_else function. The type σ in $(f, \sigma) \in \Im_{ty}$ now contains not only the arity information related to f, but also the names of the sets to which the arguments of f should belong. Now, f's arguments may be chosen from *only some* of the elements of F_{ty}, restricting the ways f may be used in a genotype. Types provide a stronger partial specification as to how f may be used than the arity does. While there is now more control over how a specific operation may be used, the extra safety is paid for by an extra layer of complexity. Assembling genotypes randomly is now a more complicated affair. Below is definition of genotypes in the typed system of representation.

Definition 3.2.3 (Genotype Definition (STGP)) Let $\Im_{ty}$ be a set of primitives. A *genotype* is now a tree in which:

- each leaf is labeled with (f, σ) where $(f, \sigma) \in F_{ty}$, and
- each inner node has two children and is labeled ϵ where ϵ is a type. The left child is either an inner node labeled $\alpha \rightarrow \epsilon$ or a leaf labeled $(f, \alpha \rightarrow \epsilon)$ and the right child is either an inner node labeled α or a leaf labeled (f, α)

Given the $\Im_{ty}$ described by table 3.2, the typed genotype depicted in figure 3.1 may be formed, and its type is $[Behavior]$. One of the advantages of this representation scheme is the new-found ability to partly specify

Type	
$[Behavior]$	User Defined Atomic Type (UDAT)
$[Direction]$	User Defined Atomic Type (UDAT)
$[Boolean]$	User Defined Atomic Type (UDAT)
Functions	**Type**
$move$	$[Direction \rightarrow Behavior]$
$turn$	$[Direction \rightarrow Behavior]$
$detectFood$	$[Direction \rightarrow Boolean]$
$if\_then\_else_{Behavior}$	$[Boolean \rightarrow Behavior \rightarrow Behavior \rightarrow Behavior]$
Terminals	**Type**
$front$	$[Direction]$
$left$	$[Direction]$
$right$	$[Direction]$

Table 3.2: A set $\Im_{ty}$ for a sample side-effect GP system capable of expressing the genotype of figure 3.1

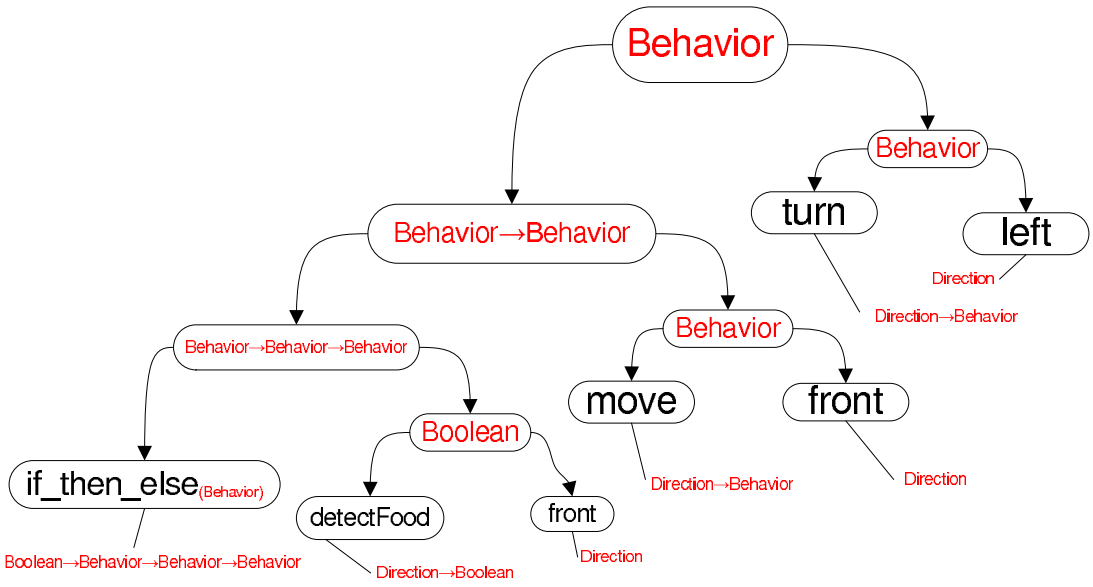

Figure 3.1: Strongly-typed genotype for a side-effect GP system based on the primitives defined in table 3.2

the type of the programs that are being evolved (for example, something that evaluates to a value of type $[Behavior]$ rather than $[Direction]$ or $[Boolean]$).

[33] applied STGP to the problem of evolving cooperation strategies in a predator prey environment. They reported that the solutions produced by a STGP system consistently outperformed the solutions produced by a standard GP system. They suggest that the reduced search space is the cause of the performance improvements. They also showed that the programs generated by STGP tend to be easier to understand. Following these encouraging results, [34] proposes the extension of STGP with a type hierarchy mechanism and describes an application to the problem of finding all the cliques in an indirected or directed graph.

3.2.3.2 Polymorphic STGP

The basic STGP formulation is equivalent to Koza's approach to constrained syntactic structures (section 3.2.1) and has the same limitation: The need to specify multiple functions which perform the same operation. For example, consider the classic example: a typed branching function if_then_else:

$$if\_then\_else : Boolean \rightarrow TYPE_1 \rightarrow TYPE_1 \rightarrow TYPE_1 \tag{3.4}$$

The if_then_else function of (3.4) is usable as a decision function between two objects of type $TYPE\_1$, but not in any other context. This implies that different versions of the same function need to be defined to handle other data types.

When a language, such as Pascal, is based on the idea that functions and procedures, and hence their operands, have a unique type, it is said to be *monomorphic*. Monomorphic programming languages may be contrasted with polymorphic languages in which some values and variables may have more than one type. *Polymorphism* is a language feature that allows values of different data types to be handled using a uniform interface. A polymorphic function is a function that can evaluate to or be applied to values of different types. A data type that can appear to be of a generalized type, for example a list with elements of arbitrary type, is a polymorphic data type. A polymorphic type system allows the expression of computations in which the types of some of the inputs (or of the outputs) are left unspecified. A polymorphic type system is necessary to express aspects of computations that behave the same independently of type.

There are two fundamentally different kinds of polymorphism, as originally described by [73]. If the range of usable types is finite and the combinations must be specified individually prior to use, it is called *Ad-hoc polymorphism*. The other, stronger kind of polymorphism, called parametric is the one that our research uses, and we will describe it in great details in the next two chapters.

Ad-hoc polymorphic functions can be applied to arguments of different types, but behave differently depending on the type of the argument to which they are applied (this is also known as function overloading). The term "ad hoc" refers to the fact that this type of polymorphism is a dispatch mechanism rather than a fundamental feature of the type system: code moving through one named function is dispatched to various other functions without having to specify the exact function being called. Overloading is simply a mechanism that allows multiple functions taking different types to be defined with the same name; the compiler or interpreter automatically calls the right one. Ad-hoc polymorphism can be generated by adding a variable mechanism to the type system, providing a way to decrease the specificity of the type included with each element of F_{ty}. STGP supports an ad-hoc form of polymorphism. Montana doesn't provide a formalization of his type system, opting instead for an informal description. He defines *generic functions* as functions which have a type that contains a variable and calls *instantiation*, the operation of applying a type to a generic function to yield a non-generic function. Montana's generic functions can only be used when a specialized primitive corresponding to the instantiation of the generic function exists. It is the specialized function that is embedded in the genotype. This is for practical reasons: directly including a parametric polymorphic object in a genotype would require a system capable of evaluating all possible instantiations of the object. Since there are an infinity of types that can be formed on any non-empty set ty, this is not feasible. Instead, Montana provides some instantiation implementations and instantiates generic functions *before* they get included in any genotype, after his system ensures that the implementation resulting from the instantiation of the variable exists. This greatly complicates the underlying system implementation (which is described on page 15 of [59]). In STGP, generic functions are defined on named lists of argument types and have their return types infered when they are embedded in a new tree not built from a crossover operation. After a generic function has been instantiated (by being embedded in a new tree) it is and behaves as a standard typed function. This is also how it is passed on to the program's descendants. Montana uses a table-lookup mechanism to produce

legal parse trees. A type possibilities table is computed beforehand to specify all the types that can possibly be generated at each tree depth level. This table provides type constraints to the function selection procedure used to generate type-correct programs. During the creation of the initial population, each parse tree is grown top-down by randomly choosing functions and terminals and verifying their validity as possible tree nodes against the type possibility table.

Generic types: Generic functions eliminate the need to specify multiple functions which perform the same operation on different types. To produce generic programs in STGP, Montana also implemented a *generic type* system. *Generic programs* are programs that have generic data types as input or output types. The generic data types are instantiated when the generic program is executed. Since generic data types can be instantiated with many different type values, the generated programs are generic programs.

Unfortunately, there are three basic limitations with STGP's implementation of polymorphism :

1. **Type possibilities table and table-lookup mechanism to instantiate type variables:** With STGP, when a parse tree is generated and a generic function is considered for inclusion, the underlying GP system must insure that the new element will make it possible to select legal subtrees. For generic functions, the GP has to loop over all ways to combine types from a type possibility table that is maintained centrally by the system.

2. **Evolving polymorphic functions and structures:** Polymorphic functions are instantiated with solid types when included in a newly spawned tree. Crossover acts on functions and terminals that are not polymorphic anymore. There is no possibility of evolving functions defined for all types. There is no possibility of evolving structures such as trees or lists.

3. **Lack of support for higher-order functions:** SGTP has no general support for growing functions which take functions as arguments and/or return functions as results.

3.2.3.3 The PolyGP system

The *PolyGP system* [14, 81, 82] is also based on a type system. Used during program creation (crossover and mutation), the type system ensures that all programs created are type-correct. In PolyGP, polymorphism is implemented through the use of three different kinds of type variables:

1. Generic type variables are used as in STGP to specify that a generated program can accept inputs of any type

2. *Dummy type variables* express the polymorphism of built-in functions: whenever they are used in a parse tree they must be instantiated to some other type (not a dummy type). If a dummy type variable occurs more than once, then it is necessary to instantiate all occurrences to the same type.

3. *Temporary type variables* are used when the type of a dummy variable can't be determined but must be instantiated to a known type. This can happen when a polymorphic function is selected to construct a program tree node.

The main differences between the PolyGP system and the STGP system are:

- PolyGP uses a type unification algorithm rather than a table-lookup mechanism to instantiate type variables.
- PolyGP uses temporary type variables to support polymorphism within a program parse tree as it is being created.
- PolyGP uses generic type variables to represent polymorphism of the generated programs. However, unlike generic data types in STGP, generic type variables are never instantiated
- PolyGP's type system supports higher-order functions (functions that take functions as arguments and/or return functions as outputs).

3.3 RECURSION AND ABSTRACTION WORK IN GP

In this section, the work related to GP and modularization is discussed. Modules encapsulate chunks of computation, and as such are a form of abstraction. Recursion is usually used in conjunction with modularization and is discussed in this section as well. There is a type of recursion used in GP called "implicit recursion" which does not need a naming or module scheme to function, and it is presented in this section as well.

3.3.1 ADFs

Automatically Defined Functions (ADFs) [49] are mechanisms devised by Koza to facilitate the creation and reuse of modules. A GP system that supports ADFs tries to discover reusable subroutines and to assemble the subroutines into genotypes. Automatic function definitions are an attempt to solve problems by automatic subproblem decomposition. The modules evolved during the system's run can be called by the programs that are being simultaneously evolved during the same run. ADF implementations rely on the use of Koza's constrained syntactic structure mechanism (section 3.2.1). It was found that genetic programs with ADFs have an advantage on some problems, particularly when the problem has a high level of regularity in its solution.

3.3.2 ADMs

[72] has proposed the use of *Automatically Defined Macros* (ADMs). An ADM is evaluated in the main program global environment, unlike an ADF where the evaluation is performed in its local environment. The idea is to simultaneous evolve programs and their control structures. When the name of an ADM is called in the main program, a macro expansion is performed. ADMs can therefore be used to implement program control structure if a block of code is passed as argument to the ADM.

3.3.3 Recursion and GP

The first efforts to evolve programs that are able to use recursion can be found on page 473 of [47] where Koza investigates a problem-specific form of recursion to solve the Fibonacci sequence induction problem. Later, [7] studied recursion use in GP in greater detail. There are currently two ways of providing recursion support in GP:

1. *Explicit recursion:* Giving names to the programs in the GP system so that they may refer to themselves

2. *Implicit recursion:* Pre-defining recursion operators in the language of the system

3.3.3.1 Recursion by naming

This approach, used in [47, 14, 7] involves giving a name to the program that is being grown. The name is included in the language of the GP, but only for that program, so that it may use it as if it were a built-in function at any point within the parse tree. This requires additional overhead. Each program now uses a slightly different language than the other programs. Names have to be kept and managed. Another problem with the recursion by naming scheme is that special mechanisms need to be put in place to handle the cases where parts of programs that refer to themselves are used to construct a new program (with a different name) in a crossover operation. The number of recursive calls must be limited to avoid infinite loops, so the number of recursive calls has to be tabulated while the program is running and a system of flags has to be implemented. In contrast, the recursion scheme of Abstraction-Based Genetic Programming does not need to provide programs with the ability to call themselves in order to support recursion. As a result of this and of the termination property of System F, it has no need to check for infinite loops.

3.3.3.2 Recursion by specialized operator

In [85], another approach for recursion in PolyGP is proposed. It is called *implicit recursion*, and it exploits PolyGP's support for higher-order functions. The idea is to implement the use of recursion using three pre-defined higher-order functions instead of explicit recursive calls. The three functions all work with lists, which are supported by PolyGP. They are:

1. *MAP:* applies the first argument, a monadic operator function which takes one argument, to each element of the second argument (a list) to produce a list of the results. In this work, we show how this can be done (using System F) without defining an external function.

2. *FOLD:* places the first argument, a function which takes two arguments, between each of the items in the list.

3. *FILTER:* applies the first argument, a predicate operator (a function which returns True or False), to each element in the second argument (a list) to produce a list containing items which satisfy the predicate operator.

Increasing the number of pre-defined functions that are manipulated by the GP system increases its search space and bloats its language with programming constructs that are not directly related to the problem that is being solved.

3.4 CONCLUSION

In this section other GP systems based on a typed representation scheme were presented. In the next chapter we will describe why we found the typed approach appealing and we list the features of the current typed representations schemes for GP that we tried to improve. Then, in section 6, we will describe our answer to

the representation issues that we will have highlighted and we will discuss the impact that our solution has on the search space of the system and in particular, we will show that our representation scheme provides an obvious, mathematically natural way to partition the GP search space that takes into account the structures of the genotypes as well as the components from which they are built.

Chapter 4

Design and Motivation

In this chapter, the motivations that lead us to propose Abstraction-Based Genetic Programming(ABGP) as a representation scheme for GP are described. Some of the contributions of our work are also introduced in this chapter. Here, we described the path that leads us to System F as the underlying language for the expression of computational structures. This description begins with the analysis of the classical representation scheme for GP (described in section 4.1) and lists the issues we encounter when we attempt to use this scheme to assemble genotypes out of building blocks. Our desire to modularize the scheme leads us to the simply typed representation scheme (section 4.2) which resolves some of the issues highlighted in the classical scheme, but creates new problems, in particular, loss of expressive power. Finally, we describe how we gain back the expressive power of the untyped system (and more) by adding an abstraction operation on types, which is in essence the novelty of this work.

4.1 Analyzing the Classical System

In this section, we study the classical GP representation scheme and identify the specific issues that motivated the formulation of Abstraction-Based Genetic Programming. We begin with a system GP_{sys} that uses a classical representation scheme (definition 2.2.3) in conjunction with a terminal set $\Im$ (definition 2.2.2).

4.1.1 Expressive Power

We wish to formulate a representation scheme for GP that is able to naturally represent the *usual data structures* independently of each system's terminal set. When we discuss the usual data structures, we are referring to the following objects:

1. Sets of constants where each constant has a data structure specific name. This includes, for example, the *Boolean* data structure (the set of two constants $\{true, false\}$). From a *structural* point of view, we ask the reader to keep in mind that the names of the constants are not important. For example, the boolean set is definable using any set of any two constants. The words "$true$" and "$false$" are only the boolean-specific terms given to these constants.

2. Data storage structures, such as pairs, lists, trees, queues, stacks and so on. These are also often associated with data structure specific named constants (the *empty_list* of lists), but also with data

structure specific named operations (the *pop* and *push* operations of a stack). These structures are often described recursively. For example, a list is either an *empty_list* (which is a constant specific to the list structure) or a *head* (list specific term) *consed* (list specific operation) with a list (called the *tail*). Note that as for the boolean example above, all that is required to define lists is a constant and an operation that associates single objects with existing lists. The actual details of the constant and of the cons operation are not relevant to the definition. This category also includes structures that are not commonly thought of as data storage structures. For example, the natural numbers, where a natural number is either the $zero$ constant or the *successor* (natural number specific operation) of a natural number. Data structure specific operations are often called *constructors* because they *build* the structure from the appropriate sub-units. For example, the pair data structure includes one (and only one) constructor, the operation that builds a pair from two objects. When constants are mixed in the structure (as they need to be for all recursive structures), the constants are also considered to be constructors (functions with 0-arity). A list has two constructors: the constant $empty\_list$ and the cons constructor. Natural numbers also have two constructors: the $zero$ constant and the $successor$ constructor.

3. Program control operations such as looping and branching. We may include these in the *usual data structure* category because when we included the last item, we included all recursive functions, but left out some of the meaning. So for example, since we never precisely defined what we meant by *successor*, we might represent *iteration* using natural numbers by changing the meaning of the *successor* operation so that it behaves as an *iteration* operation. This would allow, for example the iterative multiplication of a by b to be the $\bar{b}$ natural number where $\bar{0}$ is defined as 0 and the *successor* operation is defined as adding a to the previous number. Recursion could also be defined in terms of *iteration* (see [42]) or even in terms of lists. Similarly, by defining the Boolean set as a set of two constants, we are now free to choose any two constants, including functions. So, if we choose our two constants to be the projection operations on a pair, we can use Boolean objects directly as branching operators (in this work however, we chose a different approach for representing branching and Booleans).

Data structures often come equipped with non-constructive associated operations. For example, a list may be *folded* or *iterated* or *mapped* and a pair is associated with its two *projections*. Being able to simply define the structures is not very useful as a computational tool without the ability to also define the non-constructive operations associated with the structures, so a GP system that allows the generic representation of all the *usual data structures* must also allow for the definitions of their associated operations. The standard libraries of many modern languages include generic implementations of many data structures. However, human programmers still expect to be able to express any non-implemented usual data structure in whichever language they are working in. This expectation is founded on the underlying assumption that the *usual data structures* are the sort of abstract computational concepts that a programming language should be powerful enough to express in order to be useful. After all, a list of items is either nothing or an item linked to another (smaller) list of items. A simple pointer system is all that's needed to represent a structure of this type. Similarly, a binary tree of items is either an item or a container linking two binary trees of items. The *usual data structures* are really just specifications on how to connect chunks of data with other chunks of data, so even when the programming language lacks built-in syntax dealing specifically with list and tree structures, it is assumed that the tools needed to represent them are naturally included.

Definition 2.2.3 doesn't provide for a natural way to assemble data structures out of language primitives. The only way to increase the expressive power of GP_{sys} is to add new elements to $\Im$. Branching, for example, can only be expressed by adding the function $(if\_then\_else, 3)$ to $\Im$. Since we want to provide GP_{sys} with the ability to express all the *usual data structures*, we need to be able to express things *independently* of a genotype's terminal set. The reason for this is that we could trivially build, for any given finite subset of the *usual data structures*, a suitable terminal set that would provide the GP system with a way to express all the structures in the set.

4.1.1.1 Sensicality Issues

It is possible to include a mechanism for forming and working with lists by adding the function $(cons, 2)$ and the terminal $(empty\_list, 0)$ to $\Im$ so that the list $(cons\ x\ (cons\ y\ empty\_list))$ could be expressed as a genotype. Iteration could also be expressed by adding an iteration symbol $(iter, 2)$ allowing, for example the expression of $(iter\ var\ body)$. But doing this would imply increasing the size of $\Im$ indefinitely as we attempt to represent more and more structures. The extra elements of $\Im$ would not be directly related to the problem that is being solved, but rather to the expression of possibly irrelevant data structures. The biggest problem with this method is that we would dramatically increase the probability of constructing non-sensical genotypes. The non-deterministic nature of the recombination process of GP has no notion of context when it plugs expressions into each other. The more elements contained in $\Im$, the greater the probability of forming non-sensical genotypes. Additionally, evaluation is usually the bottle-neck of GP and larger $\Im$ sets require a heavier behind-the-scene evaluation machine. We might say that a GP system is more *expressive* than another system if it allows us to represent more with a fixed terminal set cardinality without increasing the probability of constructing non-sensical genotypes. We can state that the classical GP representation scheme doesn't allow the expression of any computation unless the necessary underlying functions and constants have been included in the terminal set and this is a problem that our research addresses.

4.1.2 Modularity

Programmers often write programs in which some programming constructs repeatedly appear. A modular GP system is a system in which each program is composed of *building blocks* or components. Large and complex structures composed of many elements are not stable in an environment where every new generation of programs is built from randomly chosen chunks of programs that exist in older generations. Recombination destroys large structures, so the ability to express strategic pieces of computation in tight packages is valuable. Building blocks are reusable anonymous closed computational units that may be combined with other blocks to form new blocks. In this context, a program is simply an assembly of blocks. Some blocks are atomic, and these represent either data or some function whose execution is outside the system. In a modular system, useful computational behavior can be extracted and reused. Implementing this requires the ability to create blocks as generalizations of existing, more specialized blocks. And of course, it also requires the ability to create a new block as a specialization of a more abstract block.

Another reason for considering a modular design is the processing power requirement of GP. Like GP, 3D programming is very processor intensive. In 3D programming however, display lists are a way to enhance

the speed of a 3D application. A display list stores a group of commands so that they can be used repeatedly just by calling the list. For instance, when modeling a car, a wheel can be defined in a display list which is then called four times with the appropriate translations. Display lists work by pre-compiling commands and storing the result in the graphics card memory. In general the execution of a display list is faster than the execution of the commands contained in it because this part of the matrix manipulation has already been done. One of our motivations for a modular design is the ability to apply the same principle to GP execution. It seemed reasonable to assume that the same approach could be used to speed up the inter-generation evaluation time of the system. Concretely, we want to be able to detect blocks that are used by several organisms so that they may be executed once, independently of the organisms that carry them. When possible, a system with this feature system would be able to store the result or the partial result of the execution of the building block with the building block. This occurs when the block can be executed independently from the many contexts in which it is used when embedded in several organisms. For example, if

$$gt\ time\ 10 \tag{4.1}$$

is a building block shared by several genotypes, it would be evaluated once (for a vector of values of $time$ if needed) and that block could be replaced directly by the result of its execution when a genotype that contains it is evaluated. Additionally, such an approach might have promising application in parallel execution research, but this is left for future work.

Consider a GP system that uses $\Im$ as defined in table 2.1. If we wanted to use $\Im$ to express a modular computation equivalent to the sentence: "If the $< time >$ variable is greater than $< 10 >$ then evaluate to $(minus)$, otherwise, evaluate to $(plus)$", we would need to be able to use the $plus$ and $minus$ terminals as leaves (or arguments to other terminals), but their arity constraint forbids this. As a result, an equivalent parse tree for this program (figure 4.1) is difficult to express using the classical scheme and practitioners use extensions to achieve modularity effects.

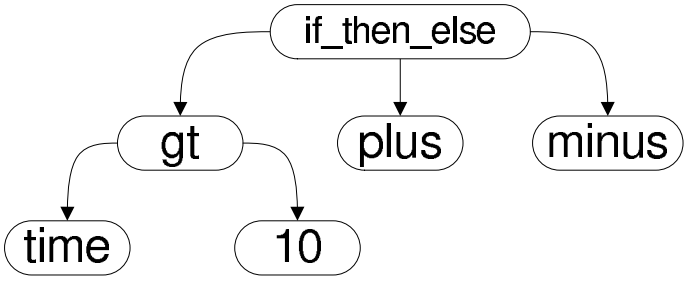

Figure 4.1: A potentially useful computational block that cannot be expressed using the classical representation scheme. The program uses the terminals of table 2.1 in a reasonable way from a computational perspective, but is an illegal formation according to definition 2.2.3

Yet, in a modular system, this type of computation could be useful as a modular arity-2 operation that could be plugged into a genotype. Because the classical scheme relies on arity as the only specification indicating how a function or a terminal may be placed in a tree, it is impossible to use any terminal with arity greater than 0 as anything else but an inner node in the parse tree. There are other issues that complicate the use of an explicit block architecture in conjunction with the classical representation scheme:

1. Non-Localized Effects of Blocks: In a block architecture, a block should be able to have non-localized effects on the whole genotype in which it is embedded. This is necessary so as to allow the representation of certain types of computations (such as recursive applications) that require such a mechanism.

2. Schema capture: [60] found that in the common GP representation, the probability of disruption of a schema changes so drastically from generation to generation that it can best be represented as a random variable. A modular GP system should allow the contraction of an aspect of a computation (even when distributed over a whole program) into a single reusable block that would be shielded from the destructive effects of crossover. This contraction should occur gradually as a response to generational selective pressure.

3. Conversely, blocks should be allowed to degrade into combinations of blocks with similar functionality but that can be affected by recombination, allowing the GP system to explore around the search space sampled by a particular block. As an example of this, consider the block $(gt\ time\ 10)$. By degrading this atomic block into a two block function application, for example $f(a)$ where $f(x) = x\ time\ 10$ and $a = gt$, it becomes possible to explore variations of this block as they naturally occur through recombination by replacing one of the two elements of the application. In later chapters, we show how this can done using λ-functions.

4. There shouldn't be any need for a central organization system that keeps track of the syntactic correctness of each combination of modules. Rather, the combination process should resemble a biological "binding" process where a block naturally binds to another block (or not) and it is easy to determine if the binding can occur, simply by looking at the two blocks. If some blocks are data, and some blocks are data structures, then following the binding analogy, data structures should naturally bind to the proper kind of data.

4.1.3 Search Space Partitioning

4.1.3.1 All Purpose GP

The issue described in this section applies specifically to value GP systems (see 2.1.1). Such systems could often be made completely reusable if they came with a mechanism that would allow the GP designer to specify the kind of programs that the system is to evolve. It is because the objective function needs to be crafted differently for each problem that these systems are not reusable in a variety of contexts. However, objective functions could be derived automatically in many instances where the system is trained on test cases. For example, suppose we took the system generated from the $\Im$ defined in table 2.1. A designer that wants to evolve programs of the form $f(x) = y$, would use an objective function computed from the number of matches of a phenotype's execution when compared to a list of test cases of the form:

$$x_0, y_0$$
$$x_1, y_1$$
$$\dots$$
$$x_n, y_n$$

A designer that wants to evolve programs of the form $f(x, y) = z$, would use an objective function computed from the number of matches of a phenotype's execution when compared to a list of test cases of the form:

$$x_0, y_0, z_0$$
$$x_1, y_1, z_1$$
$$\ldots$$
$$x_n, y_n, z_n$$

In both cases there should really be no need to rewrite any parts of the GP system's implementation. Not even the mechanisms behind the objective function's evaluation system need to be changed. All that is needed to do this is the ability to use the list of test cases *as an input to the GP system*. However, this is difficult to achieve using the classical GP representation scheme because it doesn't provide any mechanisms to restrict the search to specified partitions of the space of grammatically correct genotypes. As the zoo of structures that can be represented expands, a single system should theoretically be able to express several fundamentally different kind of programs. What is needed is a mechanism that would allow the system's owner to request (for example) the evolution of programs that take lists of numbers as input and return a single number as output, excluding other combinations, such as programs that take lists of numbers as arguments and return other lists of numbers.

A GP system that is able to represent *the usual data types* is able to use *the same* set of terminals for different problems. We would like our system to have the ability to evolve genotypes for a wide range of problems without having to change the basic underlying implementation. To do this, there needs to be a way to specify the kind of programs we want to evolve. In such a system, the set of terminals is one of the inputs that is provided to the system at the beginning of the run.

4.1.3.2 Theoretical Issues

GP schema definitions use don't care symbols ('#') to represent structural information. Program component schema theories are based on the idea of schemata as components within genotypes. This implies that schemata do not partition the search space, as a genotype may belong to several schemata. Some research has tried to correct this, such as Rosca's Rooted Tree Theorem [67] in which the definition of a schema is a rooted tree fragment. For example, the schema $(plus \ \# \ time)$ includes all genotypes whose root node is a $plus$ and whose second argument is $time$. With this schema description, a schema can be initiated at most once by a genotype and studying the propagation of the components of a schema in the population is equivalent to analyzing how the proportion of genotypes in the population that are elements of the schema changes over time. The problem is that the mathematics are complicated by the fact that the intersection of two schema may not be empty. We felt that the theoretical analysis of GP search would be simplified in a system in which there was a clear distinction between structural schemata that partition the search space into what we may call *hyperspace*, with intersecting *hyperplanes* schemata within the hyperspaces representing component inclusion.

4.1.4 Conclusion

Some principles apply when choosing a set $\Im$ for a given problem. If expressiveness is the ability of a language to express as many things as possible, then simply adding syntax to a GP system's vocabulary would make it more expressive. Moreover, if there is a positive correlation between the expressiveness of the representation scheme of a GP system and its performance, then adding syntax to the system's vocabulary should make it perform better. But that is not the case. In fact, the more syntax is added to the genetic program's vocabulary, the larger its search space and the harder it becomes for it to find a solution. What is needed is a language that contains very few symbols, from which the largest proportion as possible have a direct meaning in the problem's domain:

1. The intended meaning of the elements of the $\Im$ set should have relevance in the problem's domain. Abstract concepts, such as lists, recursion or branching have no direct concrete meaning in the problem space. Including many such terminals increases the probability of forming useless bloated structures.

2. We would like to combine elements of $\Im$ into new atomic blocks so as to express as many different complicated patterns and structures as possible. But the arity-based specification scheme of classical GP prevents us from doing this.

3. Items in the terminal set should be formulated so as to make it impossible to express programs that can not be properly evaluated by the underlying machinery of the system. The classical GP systems of today deal with sensicality issues by using more specific terminals. So for example, instead of a general $(if\_then\_else, 3)$, the more specialized functions $(if\_greater\_else, 4)$ or $(if\_detect\_food\_else, 2)$ may be used. This is because general functions offer more opportunities for the production of non-sensical computational sub-units. For example an $(iter, 2)$ symbol works well only when the first of the two arguments is a number and the second is a sub-genotype that does something.

A type system is a set of rules that specify how and when independent computational units may be combined with each other to form more complex modules. In the next section, we describe how the use of a type system resolves some of the issues carried by the classical representation scheme as outlined in this section.

4.2 Analyzing the Simple Type System

In this section, we analyze the simple time system as described in section 3.2.3.1. We begin with a system GP_{sys} that uses a simply typed representation scheme (definition 3.2.3) in conjunction with a terminal set $\Im_{ty}$ (definition 3.2.2).

4.2.1 Expressive Power

This representation scheme ensures that only sensical programs will ever be formed. However, while no non-sensical genotypes may be assembled, the number of sensical genotypes in relationship to the size of $\Im_{ty}$ that can be formed is lower than it would be with a comparably sized untyped terminal set. This is because the typing system makes it necessary to use several functions to express what could be expressed by a single function before. For example, where, branching could be expressed as a single $(if\_then\_else, 3)$ function

with the classical representation scheme, it is now necessary to define $if\_then\_else_{[Int]}$, $if\_then\_else_{[Bool]}$, $if\_then\_else_{[Int \rightarrow Int]}$, and so on, in order to do branching on each of these specific types. Just as in the classical case, this representation scheme doesn't allow the expression of any computation unless the necessary underlying functions and constants have been included in the terminal set. So if we go by the relationship between the size of $\Im_{ty}$ and what can legally be expressed by the system, this system is *less* expressive than the classical GP representation scheme.

4.2.1.1 Sensicality Issues

All sensicality issues are now resolved. A mechanism for forming and working with lists of numbers can be by adding the type [NumberList], the function $(cons, [Number \rightarrow NumberList \rightarrow NumberList])$ and the terminal $(empty\_list, [NumberList])$ to $\Im_{ty}$ so that $(cons\ x\ (cons\ y\ empty\_list))$ would be is a well-formed sub-genotype. We still need to increase the size of $\Im_{ty}$ indefinitely as we attempt to represent more and more structures, but we don't have to worry about increasing the probability of constructing non-sensical genotypes anymore. Types provide the notion of context that was missing in the untyped version, eliminating the probability of forming non-sensical genotypes. However, we haven't solved the execution problem. Larger $\Im_{ty}$ sets still require a heavier behind-the-scene execution machine and every overloaded structures and functions in the terminal set require their own execution routine. This scheme still doesn't doesn't allow the expression of any computation unless the necessary underlying functions and constants have been included in the terminal set and have their own execution routine.

4.2.2 Modularity

The simple type expression system for GP generated by definition 3.2.3 solves the classical expression problem outlined in section 4.1.2. Recall that with the classical scheme, we were unable to form the tree of figure 4.1 because we were obligated to respect arity constraints. With the typed representation, of this system (characterized by the $\Im_{ty}$ of table 3.1) extended by a terminal if_then_else of branching type $[Number \rightarrow Number \rightarrow Number]$, this issue is resolved. Figure 4.2 demonstrates this. Now, not only are we able to express the program corresponding to "If the $< time >$ variable is greater than $< 10 >$ then evaluate to $(minus)$, otherwise, evaluate to $(plus)$" as a block, but the root node of the tree (its type) provides a specification of how it may be plugged as a module inside another tree. The only remaining issue is the one described previously: there needs to exist a specific version of the if_then_else function for the type $[Number \rightarrow Number \rightarrow Number]$.

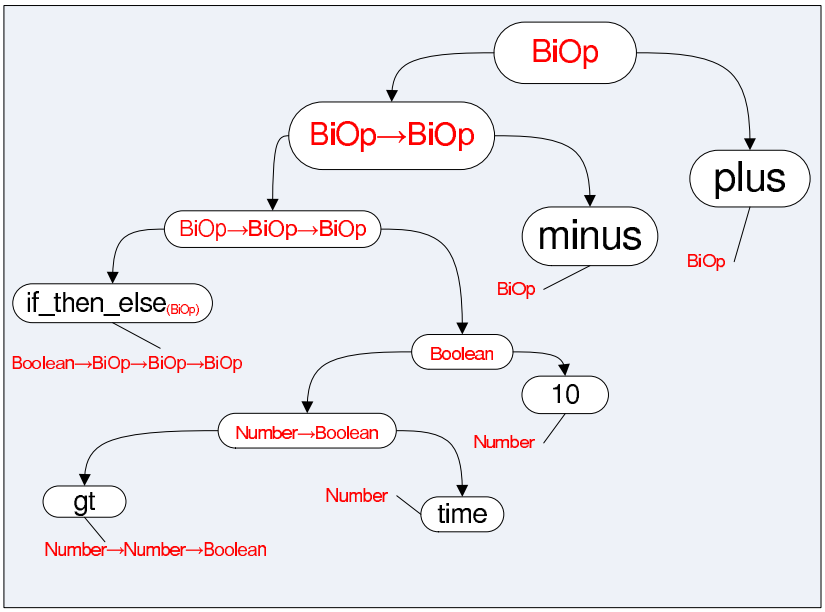

Figure 4.2: The illegal genotype of figure 4.1 can now be legally expressed using types. Note that $[BiOp]$ is a name for the type $[Number \rightarrow Number \rightarrow Number]$

.

4.2.3 Search Space Partitioning

4.2.3.1 All Purpose GP

Types can be used as the specification mechanism that we desired in 4.1.3.1. For example, a problem (and its associated objective function) can now be specified using a list of test cases in the form:

$$i_{11}^{A_1}, i_{12}^{A_2}, \ldots, i_{1n}^{A_n}, o_1^O$$

$$\ldots$$

$$i_{m1}^{A_1}, i_{m2}^{A_2}, \ldots, i_{mn}^{A_n}, o_m^O$$

where each of A_i is the type of the ith input and O is the type of the desired output. By deducing the type of the programs from the test cases, it is possible to build only programs of that type, which makes the system generic. In the case of the sample test case list above, this would mean only constructing programs of type $[A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow O]$. The remaining problem is to find a method to assemble blocks of various types into arrangements of a given type in a random manner. For this, we used the Curry-Howard isomorphism and we describe our method in the next section.

4.2.3.2 The Curry-Howard Isomorphism, Simply Typed Version

Typed GP systems are difficult to implement because the typing system generates additional problems when genotypes are initially generated by a random process (see 3.2.3.2). Each genotype must evaluate to a a specified type, and each node in the tree has to be of an appropriate type to fit in the tree. Here, we introduce a

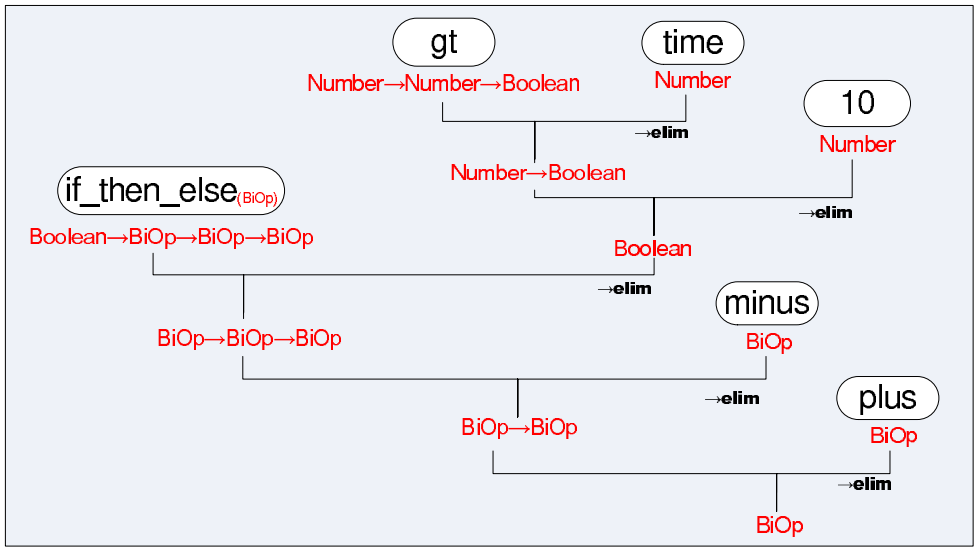

Figure 4.3: The genotype of figure 4.2 corresponds to a proof when the types of the leaves are considered to be propositions. When only simple types are considered, there is only one deduction rule, corresponding to the usual →-Elim rule

.

novel method to form genotypes from typed terminals. This method is applicable to all versions of typed GP. It is a contribution of the work presented in this thesis. It relies on the Curry-Howard correspondence [39], the generalization of which is the following claim: a proof is a program, the formula it proves is a type for the program. This can be seen as an analogy which states that the return type of a function (i.e., the type of values returned by a function) is analogous to a logical theorem, subject to hypotheses corresponding to the types of the argument values passed to the function, and that the program to compute that function is analogous to a proof of that theorem. The Curry-Howard correspondence is the observation that proof systems and models of computation are in fact structurally the same kind of objects. The isomorphism applies to typed GP systems and we were the first [6] to suggest using Curry-Howard in the context of GP. The application to the simply typed version of the GP representation scheme can be described as such: A sub-genotype composed of a left sub-genotype of type $[A \rightarrow B]$ and a right sub-genotype of type $[A]$, has type $[B]$, which corresponds exactly to an →-elim rule (i.e. if we "know" $A \rightarrow B$ and A, we can logically conclude B). For example, the proof corresponding to the the genotype of figure 4.2 is represented in tree-form in figure 4.3.

This is a significant product of this research and it solves several problems associated with typed GP systems. When we discovered the method, we called the proofs associated with genotypes via the Curry-Howard correspondence *species* as we noticed that they behave like speciation mechanisms in the biological world:

1. Species are structural patterns for genotypes
2. A genotype always belongs to one and only one species
3. Species partition the system's search space
4. Species have "entry points", the parts of the proofs that are axioms. For example the species of figure

4.3 has 6 entry points. Two genotypes that belong to the same species are different if the node corresponding to at least one of their common entry points is different in each. From this, a genotypic distance between two genotypes can be defined as the number of entry points that connect to different terminals.

Once aware of the correspondence, randomly generating viable genotypes in a typed representation scheme can be made easier by first generating proofs of an appropriate type for the problem being worked on and plugging typed computational modules into the appropriate entry points. The correspondence also induces an elegant definition of GP search space hyperspaces and hyperplanes: Hyperspaces (species) represent the structural property of a set of genotypes and hyperplanes (what's plugged into the "entry points") represent the information related to the inclusion of specific building blocks in genotypes. The intersection of any two hyperspaces is empty, but the union of all hyperspaces forms the complete search space. Species are perfect candidates for the description of search space hyperspaces as they are a representation of the specific structure of a set of genotypes, completely abstracting the components from which the genotypes that belong to the species are built. In this research, we make the following uses of species:

1. Assigning fitness values to species as the average of the fitness values of the genotypes that are instances of the species
2. Insuring that the population in each generation is genetically diverse from a structural point of view
3. As a theoretical tool to study the propagation of components and the evolution of genotype structures in a population

The first item is an extra tool that can help us tune the evolution parameters of the system more precisely in terms of helping the system identify search space subsets to explore further. As for the second item, diversity influences the rate of adaptability of the population to a dynamically changing environment by making the population explore multiple regions in the search space. Diversity also slows down the stagnation process [41]. In GP, diversity boosting strategies act by either artificially protecting specific types of individuals from death or by introducing new genetic material (by importing new individuals or generating random individuals) in the population . A diversity augmenting strategy might require the definition of a distance function between two individuals. Such a function might measure genotypic distance or phenotypic distance. Our work uses species as reliable, well-defined genotypic distances.

4.2.4 Conclusion

The classical GP paradigm uses an organism centric selection mechanism in which the organism is the unit of selection. In a gene centric evolutionary perspective, reproduction is the mechanism by which sets of genes are spread into the gene pool, species are specification for arrangements of genes, and organisms are arrangements of specific genes that assembled based on the pattern specified by the species. A typed GP system viewed in the context of the Curry-Howard isomorphism follows a very similar pattern. A gene is a typed computational block. A program is an arrangement of such blocks. The program corresponds to a constructive proof. The proof analogous to a species specifies a set of genotypes. The use of a simple type system gets us closer to our goal of building modular GP systems, but further away from our goal of increasing their expressive power. Just as for the classical case, the typed representation scheme doesn't

allow the expression of any computation unless the necessary underlying functions and constants have been included in the terminal set. Because it adds constraints on the formation of genotypes, it is now necessary to use several functions to express what could be expressed by a single function before and as a result, the ratio of sensical genotypes in relationship to the size of the terminal set that can be formed is now lower than before. The STGP generic function mechanism [59] hides this problem, but does not solve it. Generic functions are really just names for subsets of $\Im_{ty}$. This means that (for example) it is impossible to include a true if_then_else function that is able to do branching on any type. Instead, it is still necessary to define different versions of the same function to do branching on several types. It's just that now, we have a generic name for that subset of functions that we may use as well. Once the generic function is instantiated, it is and behaves like a non-generic function, and therefore requires its own symbol in $\Im_{ty}$ and its own execution mechanism. In practice, the STGP representation scheme is no more expressive than the classical representation scheme.

4.3 A Polymorphic Type System

We now turn our attention towards a stronger kind of polymorphism [73] than the one provided in the STGP specification: *parametric polymorphism*. Code is parametric polymorphic when it is written without mention of any specific type and thus may be used with any number of new types. John C. Reynolds [66] and Jean-Yves Girard [28] formally developed this notion of polymorphism, and this is how the extension to the lambda calculus known as System F came to be. Using ad-hoc polymorphism, it is possible to have a function perform two completely different things based on the type of input passed to it; this is not possible with parametric polymorphism. A parametric polymorphic function or data type is written in such a way that it may handle values identically without depending on their types [63]. Figure 4.4 demonstrates parametric polymorphism. In the figure, the symbol $pair$ is parametric-polymorphic.

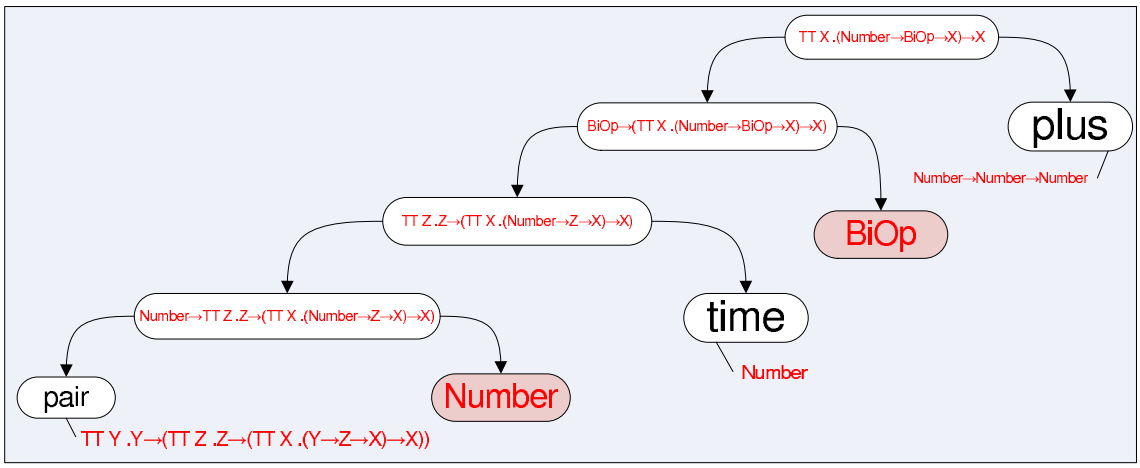

Figure 4.4: Representing a pair data structure with parametric polymorphism. This structure of the tree becomes the data structure that associates an object of type $[Number]$ with an operation of type $[Number \rightarrow Number \rightarrow Number]$

While the Abstraction-Based Genetic Programming System is the first and still the only GP system that uses a parametric-polymorphically typed representation scheme, the system presented in [81] is annotated

with types and type variables. However, that system is different from the work presented in this thesis, in which types and type variables are part of the representation. The general formalizations associated with a parametric-polymorphic system are given below.

Definition 4.3.1 (Terminal Set Definition (PPTGP)) Let ty be a set of symbols and Σ a set of type variables such that $ty \cap \Sigma = \varnothing$. Let $\mathcal{C}_{\Sigma}^{ty} = ty \cup \Sigma$. A type $[T]$ on $\mathcal{C}_{\Sigma}^{ty}$ is either an element of $\mathcal{C}_{\Sigma}^{ty}$, a construct of the form $[T_1 \rightarrow T_2]$ where T_1 and T_2 are types on $\mathcal{C}_{\Sigma}^{ty}$ or a construct of the form $[\Pi X.V]$ where $[V]$ is a type on $\mathcal{C}_{\Sigma \cup \{X\}}^{ty}$. A type on ty is a type on $\mathcal{C}_{\varnothing}^{ty}$. A set of terminals on ty, F_{ty} is a set of elements (f, σ) such that f is a symbol and σ is a type on ty. When a type σ is a type on $\mathcal{C}_{\Sigma}^{ty}$, $\mathcal{C}_{\Sigma}^{ty}$ is called the *context* of σ and the elements of ty are the *free variables* of σ.

Definition 4.3.2 (Genotype Definition (PPTGP)) Let F_{ty} be a set of terminals. A *genotype* is a tree in which:

- each leaf is labeled with (f, σ) where $(f, \sigma) \in F_{ty}$, and
- each inner node is either
  - A node labeled ϵ (where ϵ is a type on ty) that has two children, where the left child is either an inner node labeled $\alpha \rightarrow \epsilon$ or a leaf labeled $(f, \alpha \rightarrow \epsilon)$ and the right child is either an inner node labeled α or a leaf labeled (f, α)
  - A node labeled ϵ (where ϵ is a type on ty) that has two children, where the left child is either an inner node labeled $\Pi X.\sigma$ or a leaf labeled $(f, \Pi X.\sigma)$ and the right child a leaf labeled α and replacing all occurrences of X by α in σ yields ϵ.

Using this definition, a generic function version of the if_then_else function can be re-included in the terminal set defined in table 3.2 by adding the terminal $(if\_then\_else, [Boolean \rightarrow (\Pi X.X \rightarrow X \rightarrow X)])$.
Figure 4.5 illustrates an example of such a polymorphic branching function; in this case, the result of the application of $(if\_then\_else\ (detectFood\ front))$ branching is combined with the $[Behavior]$ type to form a sub-tree of type $[Behavior \rightarrow Behavior \rightarrow Behavior]$ which is exactly the type of a decision function that chooses between two objects of type $[Behavior]$ following the result of the evaluation of Boolean predicate. The Curry-Howard isomorphism is extended to this type system by adding a Π-elim rule, the equivalent of a second-order $\forall$ elimination rule.

The typing information complicates the graphical tree notation. For the purpose of simplifying our genotype notation, we now introduce an alternate equivalent representation styles for polymorphically-typed genotypes. Simplified Tree Notation (STN) is a shorthand tree form notation. Since we can always deduce the type of an inner-node from the type of its children, we may forego the labeling of the inner nodes completely (except when it adds to the clarity if the genotype). By convention, we label at least the type of the root of the genotype to indicate what it evaluates to. Illustrating an example of this tree notation, figure 4.6 rewrites the genotype of figure 4.4 in this style.

4.3.1 Expressive Power

There are surprising representation effects that result from the use of parametric polymorphism. For example, we quickly notice that not only can we now easily express a function such as if_then_else by giving it the

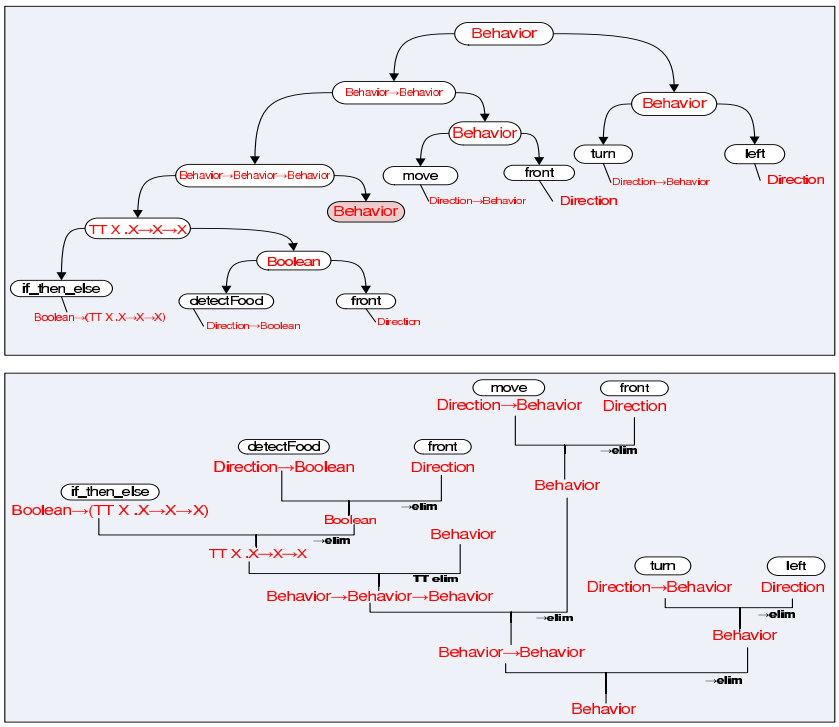

Figure 4.5: The parse tree of a polymorphically typed genotype (above) and its corresponding Curry-Howard proof tree (below)

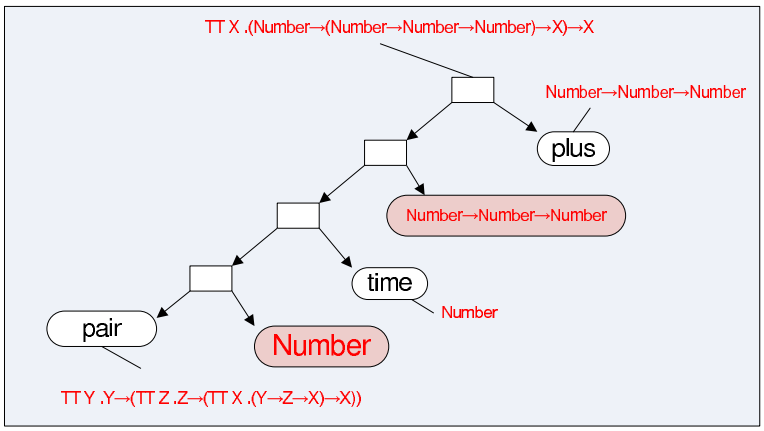

Figure 4.6: Rewrite of the genotype of figure 4.4 in the STN style

type $[Boolean \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$, we don't even need to express it anymore. By replacing the $[Boolean]$ atomic type by the abstract type $[\Pi X.X \rightarrow X \rightarrow X]$, it is now possible to express a decision without any specialized decision operator! Figure 4.7 illustrates how this clever use of polymorphic typing allows such a thing to happen. By typing the *gt* function in such a way that it returns a result of type $[\Pi X.X \rightarrow X \rightarrow X]$, we remove the need to include the [Boolean] type as a user defined atomic object. We are now working directly with the *structural description* of the [Boolean] type: The type of a set built from two constants. Table 4.3.1 illustrates what a terminal set (which we will refer to as a *context*) looks like when

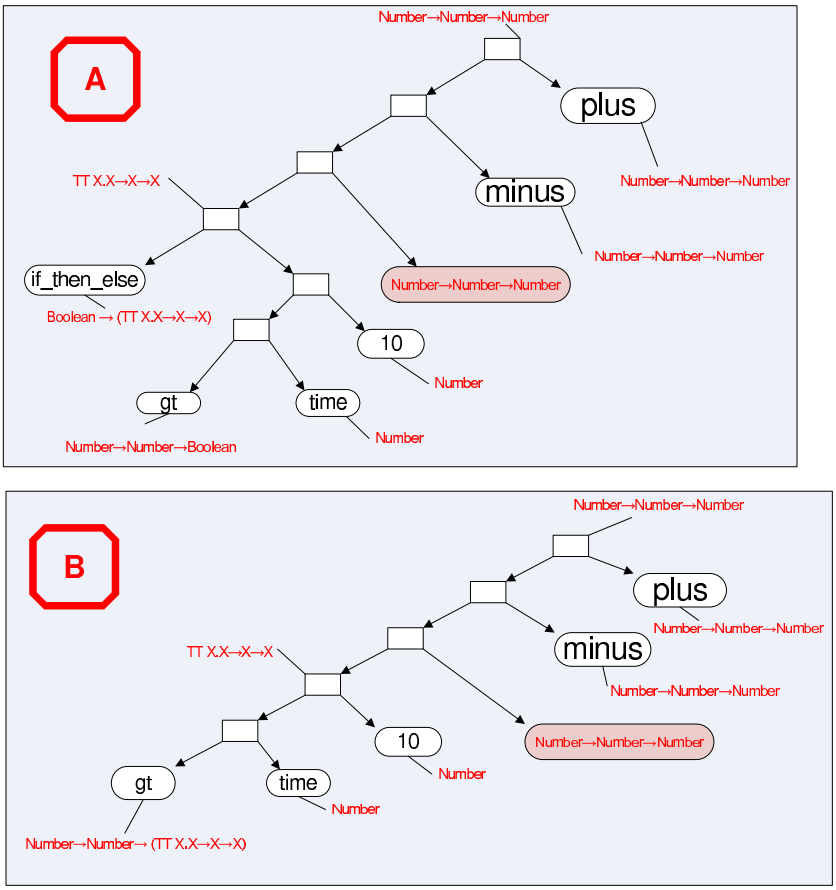

Figure 4.7: Two polymorphically-typed versions of the computational block represented in figure 4.2. Because the gt operator of the terminal set of the genotype labeled B evaluates directly to an appropriate projection function, the genotype is able to express branching without a branching operator.

it uses polymorphic typing. It contains less elements than the previous version, but can still express all that could be expressed before. Now, all terminals are strictly related to the problem logic, and there are no more structures related to the program logic. This means that we may model more complex problem environments without over-expanding the system's $\Im_{ty}$set. As it turns out, this representation scheme allows us to do what we set out to do at the beginning of this chapter: represent the usual data types. We now cover a few examples to demonstrate the additional expressive power that results from the added type abstraction mechanism.

context 4.3.1 Types, functions and terminals definition for polymorphically-typed version of the symbolic regression example

Type	
Number	User Defined Atomic Type (UDAT)
Functions	**Type**
$plus$	$[Number \rightarrow Number \rightarrow Number]$
$minus$	$[Number \rightarrow Number \rightarrow Number]$
div	$[Number \rightarrow Number \rightarrow Number]$
$times$	$[Number \rightarrow Number \rightarrow Number]$
gt	$[Number \rightarrow Number \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$
Terminals	**Type**
$[0, \ldots, 10]$	$[Number]$
time	$[Number]$

4.3.2 Expressing Unions

Consider a GP system with its $\Im_{ty}$ defined by context 4.3.2. In computer science, a union is a data structure that stores one of several types of data at a single location. Our first data structure will be a union that represents a unit of space in the environment modeled by the $\Im_{ty}$ defined by Table 4.3.2. In the problem logic, we might suppose such a structure to be a store for one and only one of:

1. an object of type $[Food]$
2. an object of type $[Pheromone]$
3. nothing (type $[Nothing]$)

The type of such an object is $[\Pi X.(Nothing \rightarrow X) \rightarrow (Food \rightarrow X) \rightarrow (Pheromone \rightarrow X) \rightarrow X]$. In order to use such a structure, for example as a an argument to a predicate, an object of this type would need to be connected to three functions, namely:

1. an object of type $[Food \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$
2. an object of type $[Pheromone \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$
3. an object of type $[Nothing \rightarrow (\Pi X.X \rightarrow X \rightarrow X)])$

The object itself would then be responsible for applying the correct predicate depending on its contents.

4.3.3 Expressing Pairs

Adding a $(pair, [\Pi Y.Y \rightarrow (\Pi Z.Z \rightarrow (\Pi X.(Y \rightarrow Z \rightarrow X) \rightarrow X))])$ terminal to the terminal set defined in table 4.3.1 allows the representation of pair structures. This is the structure illustrated by figure 4.6.

context 4.3.2 A context that provides the appropriate definitions for a sample side-effect polymorphically-typed GP system

Type	
$[Behavior]$	User Defined Atomic Type (UDAT)
$[Nothing]$	User Defined Atomic Type (UDAT)
$[Food]$	User Defined Atomic Type (UDAT)
$[Pheromone]$	User Defined Atomic Type (UDAT)
$[Direction]$	User Defined Atomic Type (UDAT)
Functions	**Type**
$move$	$[Direction \rightarrow Behavior]$
$turn$	$[Direction \rightarrow Behavior]$
$detectFood$	$[Direction \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$
$detectPheromone$	$[Direction \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$
Terminals	**Type**
$empty$	$[Nothing]$
$front$	$[Direction]$
$left$	$[Direction]$
$right$	$[Direction]$

4.3.4 Expressing Lists

If we extend context 4.3.1 with the objects defined in context 4.3.3, we gain the ability to form tree fragments that follow a specific regular pattern (see figure 4.8). This structure behaves like list of numbers telling us that a polymorphic list specialized on the type $[Number]$ has type $[\Pi X.X \rightarrow (Number \rightarrow X \rightarrow X) \rightarrow X]$. The elements of Table 4.3.3 can be added to any terminal set, giving it the ability to represent lists. The power

context 4.3.3 Terminals and function a list structure extension

Functions	**Type**
$cons$	$[\Pi Y.Y \rightarrow (\Pi X.X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X) \rightarrow (\Pi X.X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X)]$
Terminals	**Type**
$empty\_list$	$[\Pi Y.\Pi X.X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X]$

of this representation method is illustrated by the next example. If we wanted to represent an operation on the list, such as the recursive addition of all the elements, resulting in an object of type $[Number]$ *we wouldn't need to add any other terminals* to the system's $\Im_{ty}$ set. Figure 4.9 illustrates this. Our typing system ensures that there is no ambiguity as to what we mean to do. This is a new scheme for recursion representation in GP, and it is a novel contribution of our research.

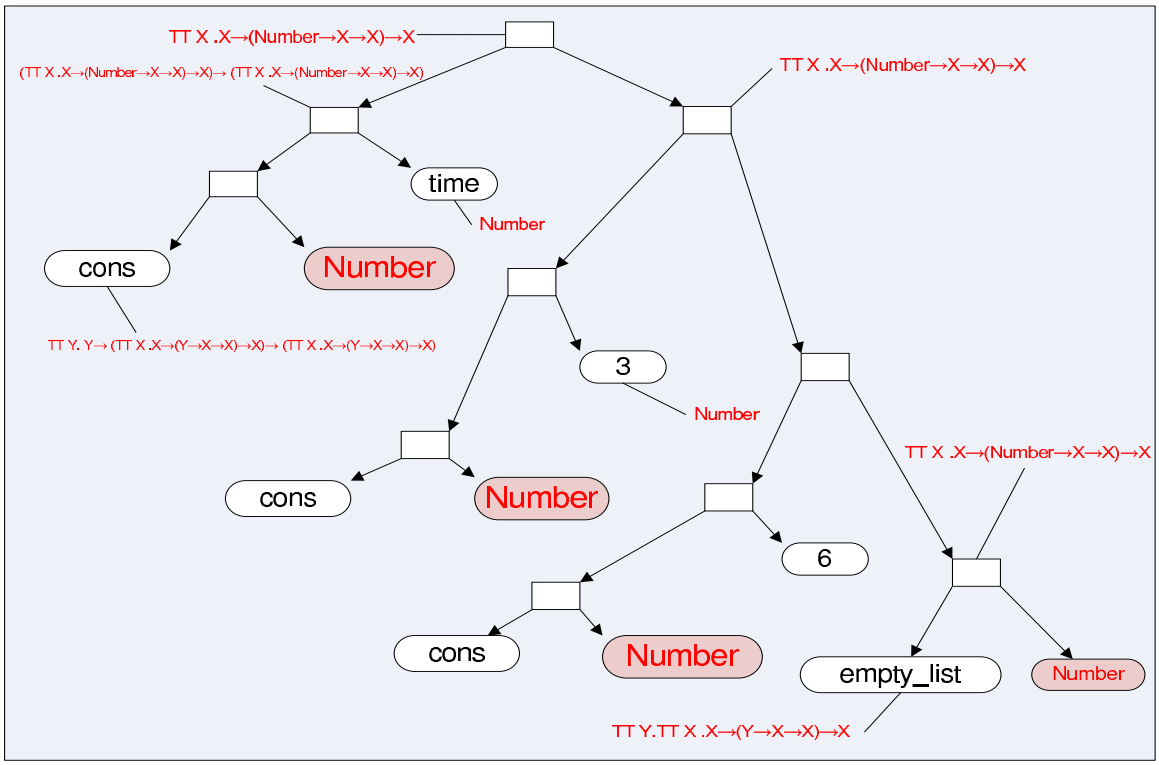

Figure 4.8: A list of numbers represented using parametric list constructors

4.3.5 Conclusion

There are still ordinary computations that this representation scheme does not allow. For example, we could easily include an iteration procedure on a predicate by adding $(while, [\Pi X.Bool \rightarrow X \rightarrow X])$ to the system (figure 4.10 demonstrates this), but we wouldn't be able to guarantee that the program ever terminates. The strength of the representation comes from the execution procedure that we describe in the next section. Type systems alone are not sufficient to get us closer to our goal of being able to express all the *usual data types* in a safe manner. We need polymorphic typing to indicate that parts of a function are not type specific, but we also need a way to express the parts of a function that are not type specific in a non type specific manner. It is at this stage of the research process that we realized that to include functions that are truly polymorphic in our genotypes would necessitate the ability to express functions that behave the same independently of type considerations. Moreover, we need to know that the genotypes express programs that actually terminate.

4.4 Motivation for Using a Lambda Calculus

We have stated that our research is motivated by the need to formulate a representation scheme for GP with the ability to naturally represent the *usual data structures* independently of each system's terminal set. In this section, we describe the role that functional abstraction holds in this context and why functional abstraction mechanisms are necessary for our representation scheme. We use λ-abstractions to express functional abstraction, and the precise mechanisms for doing so are described here.

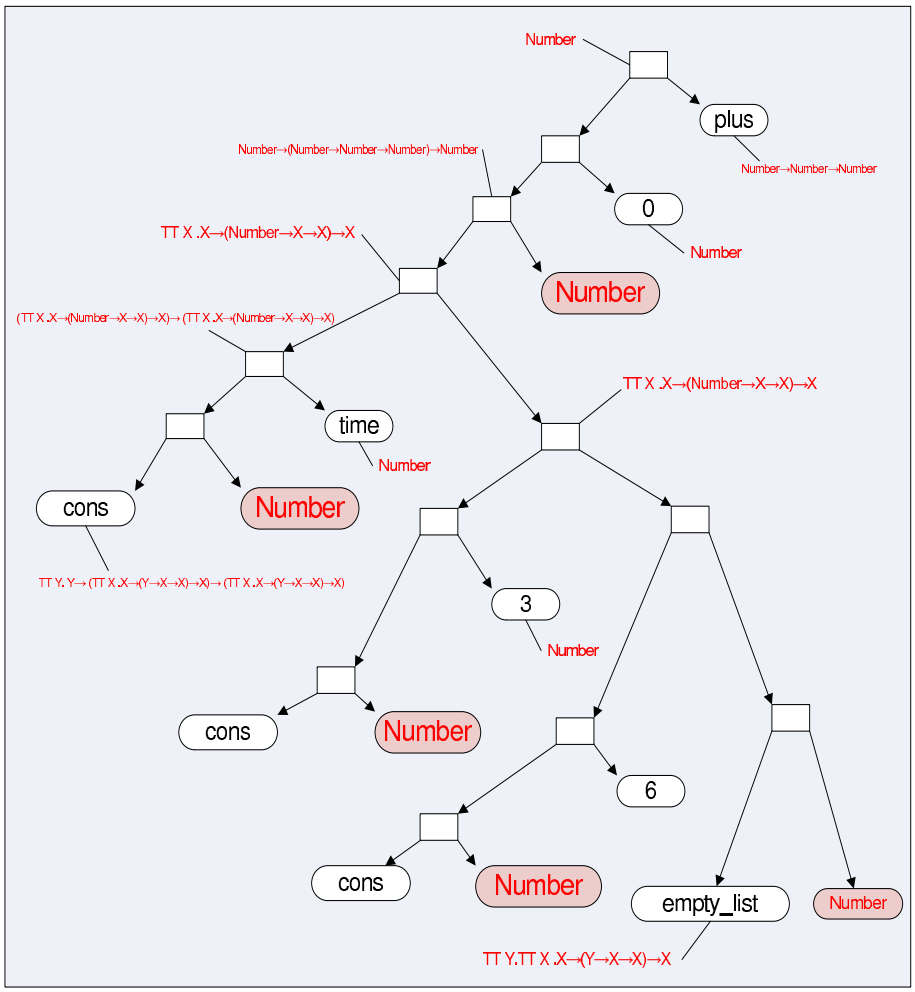

Figure 4.9: A genotype that represents a program that adds all the elements of a list of numbers

4.4.1 Adding Functional Abstraction Mechanisms

Syntactic typing restricts the form of the resulting program, but *does not allow* higher level structures to be represented ([76, 82, 83]). In STGP, the implementation of a type abstraction mechanism used by Montana is restricted by the need to appropriately overload all versions of a generic function so that it may work in any context in which it is used [34]. We already know that this is impossible in practice. The remaining problem concerning the use of a truly parametric polymorphically-typed system for GP can be stated as: Given a parametric polymorphically typed genotype, how does it get executed?

In the parametric-polymorphic specification, when a function $(f, [\Pi X.V])$ is included in the terminal set of the system, it may be used with any type $[B]$, which means that there must be an evaluation procedure for all possible sub-genotypes $(f\ [B])$ of type $[V[B/X]]$. The solution is to make sure that within the evaluation procedure of f, no specific information about the type $[X]$ is needed. The function f has its context, and within it, the type $[X]$ is treated as an atomic type. The typing system described here is of course the typing system of System F. This is exactly what this motivation section leads us to. A polymorphically-typed lambda

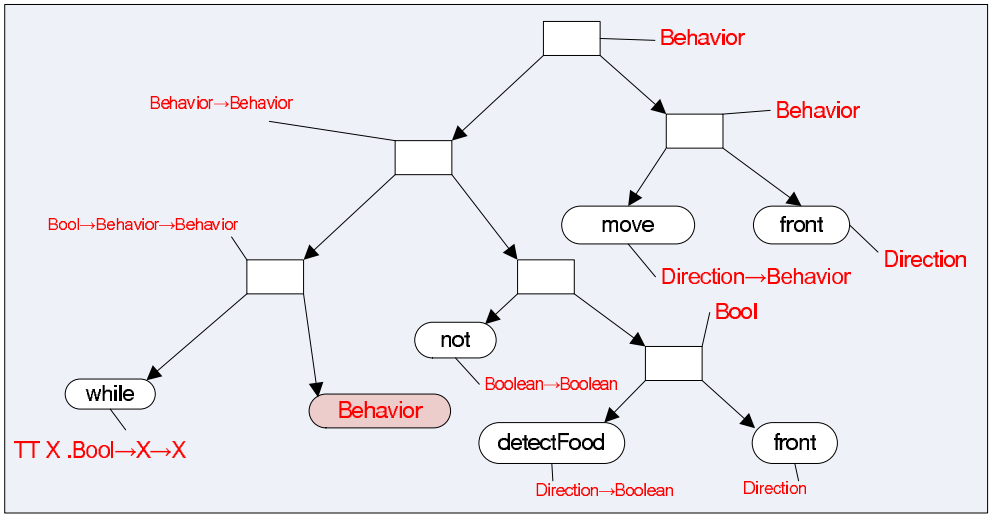

Figure 4.10: Polymorphically-typed looping construct

calculus is the perfect compromise between being able to express many things, while retaining a small sets of primitives. Parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety. Typically, when a genotype is executed, the system's $\chi_{(Ph)}$ function has some (possibly recursive) execution component for each symbol in the system's terminal set. Each component is linked to some system internal execution routine that does the computation that is represented by the symbol and returns a value or applies a side effect. For an example of this, please see table 2.2 which describes the components of two execution functions on the same terminal set. It is not possible to do this with a symbol that represents a parametric polymorphic function because we still don't know how to code functions that have computational parts that are independent of type. We will describe how we can code such functions using System F in the next chapter, but in the meantime, we may still describe the behavior that the execution path of such a system should take without any knowing anything about System F's representation scheme. To do this, we'll use the set $\Im_{ty}$ defined by Table 4.3.2 to create the genotype 4.2.

$$\begin{aligned}&((detectFood\ front)\ [Behavior]\\&\quad(move\ front)\\&\quad(turn\ ((detectFood\ left)\ [Direction]\ left\ right)))\end{aligned} \tag{4.2}$$

The parse tree corresponding to this program is depicted in figure 4.11. When the system's $\chi_{(Ph)}$ function executes this program, there may be several entry points. Suppose that our $\chi_{(Ph)}$ function begins its execution by the evaluation of the sub-term $(detectFood\ left)$, in this case, assuming that there is no food to the left of the organism. Rather than returning a value, the execution function in this case produces a new tree and replace the original genotype by this new partially evaluated tree. The tree resulting from this partial execution is depicted in the first part of figure 4.12. The same process is repeated when the execution function comes to the evaluation of the $((detectFood\ front)\ [Behavior])$ subtree (part 2 of figure 4.12). Assuming that there is food in front of the organism, what must happen is that the execution of some of the genotype's subcomponents must result in a new partially evaluated genotype rather than in a typed value used by a recursive execution procedure.

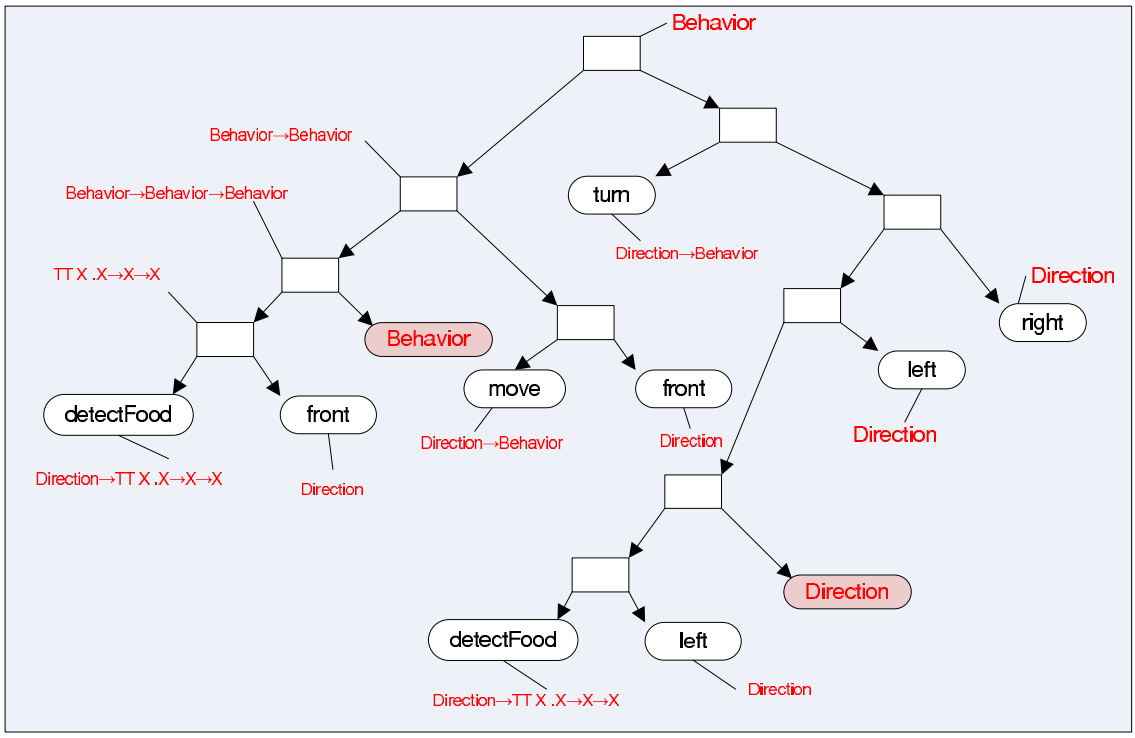

Figure 4.11: The STN parse tree corresponding to term 4.2

4.4.2 System F

This leads us to the use of System F as a representation scheme. In System F, types are explicit *in* the computation. Unlike other type systems, in System F, terms interact directly with types. Some terms are functions that take types as parameters and return other functions (such as $\Lambda X.v$). The ability to express functions that take types as parameters gives the ability to express programs that behave the same on any argument type. For example a program of type $[\Pi X.X \rightarrow (X \rightarrow X) \rightarrow X]$ would see its System F encoding roughly translated in English as: "Take a type X as argument, followed by a variable x of type X and a function f of type $[X \rightarrow X]$. Return the value $f(f(x))$". Once we settled on System F as a primary representation scheme, we realized that it would be well suited to a GP system in which the programs were assembled from modular components. This step is also described in detail in the next chapter.

4.5 Conclusion

In the next chapter, we will describe System F and show how its expressive power allows it to represent all provably total functions, which means that a GP system that uses a System F-based representation scheme always has, at least, the capacity to represent the great majority of usual structures that are typically used in programming independently of its terminal set. We will show how the use of System F decentralizes the process of type checking during tree generation and how it is always possible to execute a tree resulting from the combination of two System F based components, even when one of the component is polymorphic, eliminating the need for types possibility tables.

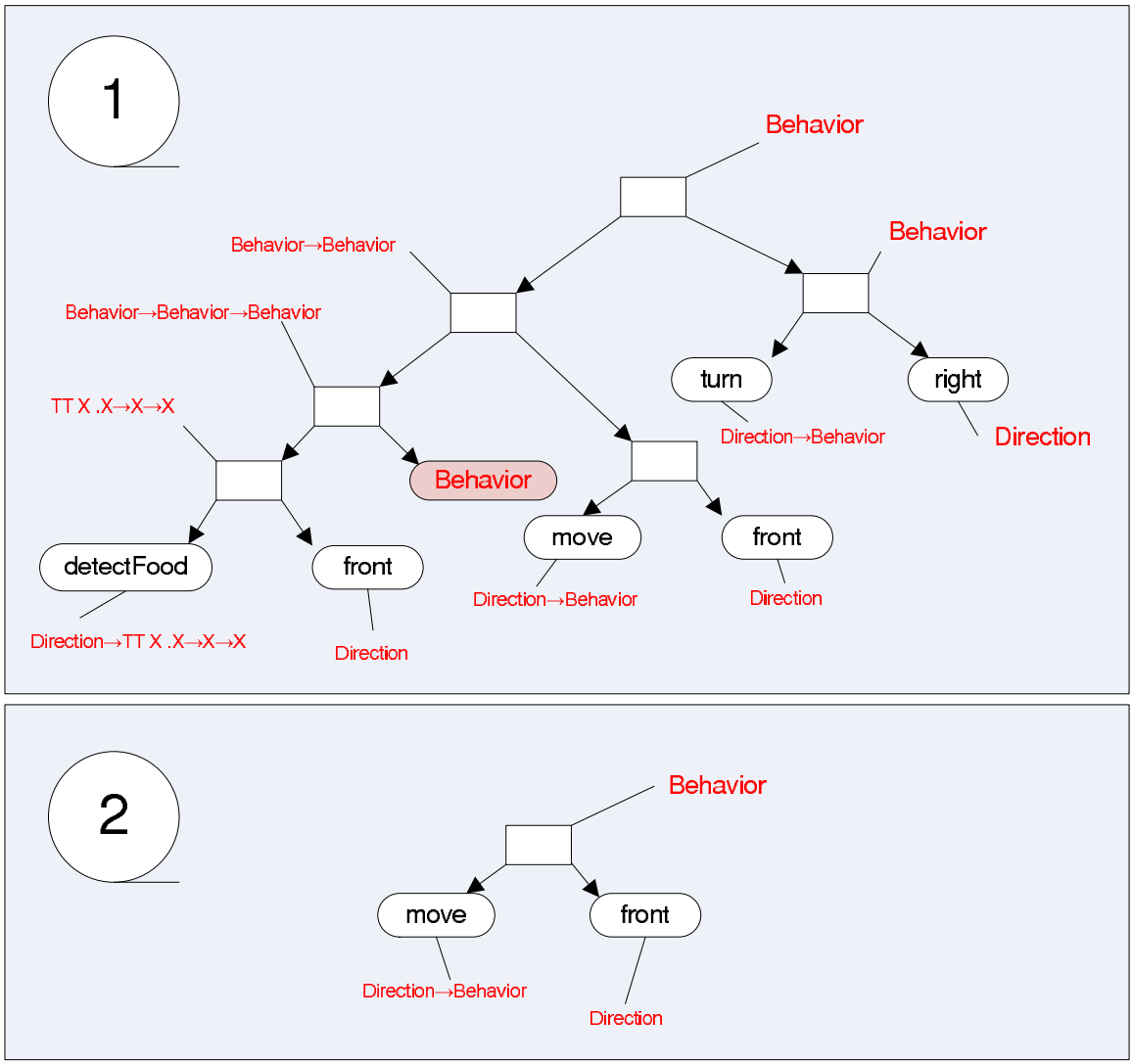

Figure 4.12: Execution sequence of the genotype of figure 4.2

Chapter 5

System F

The next chapter describes Abstraction-Based Genetic Programming(ABGP). The linguistic foundation of ABGP is System F, also known as *polymorphic λ-calculus* or *second-order λ-calculus* which we describe in this section. It is a typed λ-calculus discovered independently by the logician Jean-Yves Girard [28, 42] and the computer scientist John C. Reynolds [66]. The calculus is *impredicative* making self-application possible; for example, the polymorphic identity $(\Lambda X.\lambda x^X.x)$ can take its own type,$[\Pi X.X \to X]$, and then itself as arguments. The definition of an object can refer to the collection to which the object belongs. System F renders all the usual data-types definable as pure abstractions and directly supports recursion without naming or special operators. In this chapter, we present System F and we lay the theoretical foundation for the next chapter which gives the relevant definitions (terminal set, genotype) for the Abstraction-Based Genetic Programming System, a parametric-polymorphically typed GP system that uses System F to express its genotypes.

Notation: We write $E\{\epsilon := \beta\}$ to indicate that the expression β replaces all instances of the variable ϵ in the expression E. When we replace a variable in an expression, we assume that all issues of bound variable capture have been resolved, for example by appropriate renaming (see 5.1 for the related concepts of α-conversion and α-equivalence).

5.1 λ-CALCULI

A λ-calculus is a formalism that uses the symbol λ to denote anonymous function abstraction. Originally conceived by Church [11], the first λ-calculus turned out to be inconsistent [44], but the subsystem dealing with functions became a successful model for the computable functions. Representing computable functions as expressions in a λ-calculus gives rise to what is called *functional programming* [4].

α-Conversion and α-Equivalence α-conversion is a bound variable renaming operation. It expresses the notion that the names of the bound variables are unimportant; Two expressions are *α-equivalent* when there is an α-conversion sequence from one to the other. The meaning of α-conversion is the same for all λ-calculi, but the details differ depending on the system that is being studied.

5.1.1 The Type-free λ-calculus

In the untyped λ-calculus, every expression considered as a function may be applied to every other expression considered as an argument. For example the identity function $\lambda x.x$ may be applied to any argument x to give as result that same x. In particular, it may be applied to itself.

5.1.1.1 Application and Abstraction

The calculus has two basic operations:

1. *Application*, in which the expression FA denotes the data F considered as algorithm applied to A considered as input. Expressions such as FF, where F is applied to itself, are allowed. This permits the simulation of recursion.

2. *Abstraction*, in which the expression $\lambda x.M$ denotes the intuitive map that assigns M to a variable x. x may appear in M as a *bound variable*.

The principle axiom scheme of the λ-calculus is the β-rule:

$$(\lambda x.M)N = M\{x := N\} \tag{5.1}$$

5.1.1.2 α-Conversion and α-Equivalence for the Type-free Calculus

α-conversion for the type-free calculus is limited to bound variable renaming in the terms. For example, the two terms $(\lambda x.x)$ and $(\lambda y.y)$ are α-equivalent.

5.1.1.3 λ definability

In the lambda calculus one can define numerals and represent numeric functions, such as the functions plus, times and exponentiation. In fact, all computable recursive functions can be represented in the type-free λ-calculus (see [35] and [4]).

5.1.2 Typed λ-calculi

Typed versions of the λ-calculus were introduced in [17] and [12]. The two original papers of Curry and Church introducing typed versions of the λ-calculus give rise to two different families of systems.

1. In the Curry style, typing is implicit and terms are those of the type-free theory. Each term has a (possibly empty) set of possible types. A program may be written without typing at all, leaving it to the compiler to check whether a type can be assigned to the program. This will be the case if the program is correct. A well known example of a programming language that uses this style is ML [56].

2. In the Church style, typing is explicit and the terms are annotated versions of the type-free terms. Each term has a type that is usually unique up to α-equivalence and that type is derivable from the way the term is annotated. Here a program should be written together with its type. For these languages typechecking is usually easier since no types have to be constructed. Examples of languages that uses this style are ALGOL and PASCAL.

we use the Church style formulation in which types are embedded in terms, resulting in an extension of the pure λ-calculus. Types provide a partial specification of the algorithms that are represented and are useful for showing partial correctness. They may also be used to improve the efficiency of compilation of terms representing functional algorithms.

5.1.2.1 α-Conversion and α-Equivalence for the Simply-Typed λ-calculus

Here, α-conversion is also limited to functional abstractions, but within type constraints. For example, the two terms $(\lambda x^U.x)$ and $(\lambda y^U.y)$ are α-equivalent , but $(\lambda x^U.x)$ and $(\lambda y^V.y)$ are not.

5.1.3 Expressivity

The simply-typed λ-calculus has very little expressive power and is too restricted for our purposes. For example, it can't be used to express all the computable functions that are needed in programming, and it can't be used to represent integers or booleans, and their associated functions. Other systems have therefore been proposed to obtain more expressivity. For example, Gödel's system T [24], which mixes primitive recursion and the simply-typed λ-calculus. In system T, it is possible, for example, to express non-primitive recursive functions such as Ackermann's function. The problems are that, as [42] puts it (pg 46):

- Systems like T are a step backwards from the logical viewpoint: the new schemes do not correspond to proofs in an extended logical system [...]
- By proposing improvements of expressivity, these systems suggest the possibility of further improvements. For example, it is well known that the language PASCAL does not have the type of lists built in! So we are led to endless improvement, in order to be able to consider, besides the booleans, the integers, lists, trees, etc. Of course, all this is done to the detriment of conceptual simplicity and modularity.

The second problem, in particular, resonates with our motivations for using a λ-calculus as a scheme to express genotypes in a GP context. It is also possible to increase the expressivity of the calculus by allowing quantification on type variables. This is the method that System F uses. System F is obtained by adding an operation of abstraction on types. This operation is extremely powerful and in particular all the usual data-types (integers, lists, etc.) are definable [42]. System F is introduced in the next section (5.1.4)

5.1.4 System F

The expressions of the system are built using the following grammar:

Definition 5.1.1 (System F Grammar)

```
<Type> ::=  | C
            | X
            | <Type> -> <Type>
            | TT X . <Type>

<Term> ::=  | C
            | x
            | <Term> <Term>
            | "lam" x . <Term>
            | "Lam" X . <Term>
            | <Term> <Type>
```

Notation: Within the text, we use the Greek letters Π, λ and Λ to represent the strings "`TT`", "`lam`" and "`Lam`" respectively. We write either $y \ : \ [T]$ or $y^{[T]}$ to indicate that (y) is a term of type $[T]$. Finally, we write $name \stackrel{\text{def}}{=} E$ when we want to use $name$ as a shorthand for the expression E.

Well-typed terms of System F cannot be described fully by a context free grammar. Section 5.2 describes type constraints and section 5.3 describes term constraints.

5.2 TYPES IN SYSTEM F

The most basic combination rule is the rule that allows a function f of type $[A \rightarrow B]$ to be applied to an object x of type $[A]$, yielding the object $(f\ x)$ of type $[B]$.

In a modular programming environment where closed independent modules are combined with other modules, some functions are more useful when the types of some (or all) of their arguments are left unspecified. Our classic example is an if_then_else function which expresses the computation: "take an argument a of type $[Boolean]$ and two arguments b and c of unspecified type $[X]$. Evaluate to b if a evaluates to $true$ and evaluate to c otherwise". By not specifying types for b and c, the function can be used as a branching module in any context. The only problem is that in order to use the if_then_else function as a module in a type system, it also needs to itself have a type. System F provides a simple and elegant solution by making it possible for functions to take types as arguments. For example, if_then_else can be expressed in System F by a term equivalent to the sentence: "Take a type $[X]$ as your first argument, then an argument a of type $[Boolean]$ and two arguments b and c of type $[X]$. Evaluate to b if a evaluates to $true$ and evaluate to c otherwise". This is an *abstraction*, a term that expresses behavior independently of the types of some (or all) of the arguments that it may be applied to. Another example is the term 5.2:

$$(\Lambda X.\lambda x^X.\lambda y^X.x) \tag{5.2}$$

which expresses the program: "Take a type $[X]$ as argument and two arguments of type $[X]$ and return the first of the two arguments". This program works the same, independently of the type with which it is used.

A type can be a *type constant* (for example, the type $[Int]$), a *type variable* or a composition of types assembled by the following 2 rules:

1. If $[U]$ and $[V]$ are types, then $[U \rightarrow V]$ is the type of a function that takes an argument of type $[U]$ and returns a result of type $[V]$.

2. If $[V]$ is a type, and X is a type variable, then $[\Pi X.V]$ is a type and X is bound in $[V]$. The names of the bound variables are unimportant as long as we remain consistent; for example $[\Pi X.X \rightarrow X]$ and $[\Pi Y.Y \rightarrow Y]$ are the same type.

The function $plus$, an example of a constant term, takes two arguments of type $[Int]$ and returns a result of type $[Int]$. The type of this function is $[Int \rightarrow Int \rightarrow Int]$. The $\rightarrow$ operator is binary, and gives precedence to the left. This is important because it affects the system's behavior. For the $plus$ example, $[Int \rightarrow Int \rightarrow Int]$ types a function that takes an object of type $[Int]$ and returns a *function* of type $[Int \rightarrow Int]$. The only objects that can be plugged into $plus$ are objects of type $[Int]$. In this thesis, terms are enclosed in parentheses, so assuming (5) is term of type $[Int]$, then $(plus\ 5)$ is a well-formed term with a meaning of its own as a function of type $[Int \rightarrow Int]$. Precedence for terms is to the right, so $(plus\ 5\ 5)$ is equivalent to $((plus\ 5)\ 5)$. The type $[\Pi X.V]$ is called an *abstract type*, and an object of this type can be expressed by a term in the general form:

$$\Lambda X.(function\_body) \tag{5.3}$$

Where X is understood to be the same type variable as the one in the type of the object. The calculus requires that any variable of type $[X]$ within $function\_body$ be bound by a λ-abstraction. A function typed with an abstract type $[\Pi X.V]$ can be viewed as a function which takes as its first parameter a type $[Ty]$ and returns a function typed as $[V\{X := [Ty]\}]$.

5.2.1 System F's Curry-Howard Isomorphism

Considering terms and types as programs and their specifications is not the only possibility. In section 4.2.3.2 of our Motivation chapter, we've described how passing an object of type $[A]$ to a function of type $[A \rightarrow B]$ produces an object of type $[B]$, which is exactly equivalent to an $\rightarrow$-elim rule. Then, in 4.3, we've described how a type such as $[\Pi X.X \rightarrow X \rightarrow X]$ may also be read as the second-order proposition $\forall X.X \rightarrow X \rightarrow X$. Passing a type $[A]$ as an argument to an object of type $[\Pi X.V]$ produces an object of type $[V\{X := A\}]$ which is exactly equivalent to a second-order Π-elim rule.

The idea that a type [A] can also be viewed as a proposition and a term of [A] as a proof of this proposition is independently due to de Bruijn [22] and Howard [40]. Both papers were conceived in 1968, but hints to this propositions-as-types interpretation were given as early as 1958 in [18] and in [54]. Several systems of proof checking are now based on this interpretation of propositions as types and of proofs as terms. We make extensive use of the correspondence in this research.

5.3 TERMS IN SYSTEM F

Terms are the computational modules of the system. A term is a program when it can be reduced to another term and a value, or *normal form* when it can't. Reduction is the operation that replaces all occurrences of a bound variable in the context in which it is bound. Normalization is done by the application of successive reduction operations to a term until it can't be reduced further. Valid terms are defined below. For each case, we recall the corresponding Simplified Tree Notation (STN) as presented in section 4.3.

1. *constants and variables:* These are typed names/symbols such as 5 of type $[Int]$ or $sugar$ of type $[Food]$. We often write (y^T) to indicate that y is a variable of type $[T]$. In STN form, constants are leaves.

2. *functional applications:* If (f) is a term of type $[U \rightarrow V]$ and (a) is a term of type $[U]$ then $(f\ a)$ is an application of type $[V]$. Using STN, we represent the functional application $(f\ a)$ of type $[A]$ by a tree with root node labeled *A*, with its left subtree representing the term (f) and its right subtree representing the term (a). Figure 5.1 represents the term $(plus\ 5\ 5)$ in its STN form.

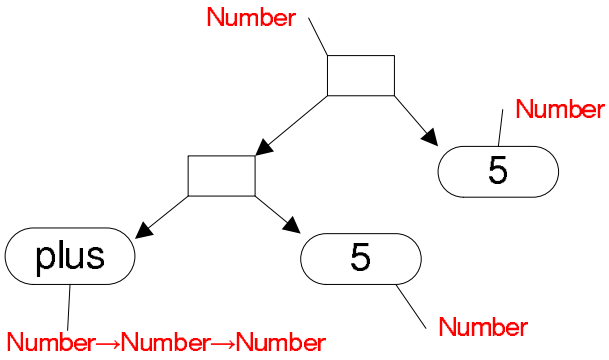

Figure 5.1: Parse tree for term $(plus\ 5\ 5)$

3. *λ-abstractions:* Assuming that (v) is a term of type $[V]$, then $(\lambda x^U.v)$ is a term of type $[U \rightarrow V]$. The variable x^U is bound in (v) by the λ operator. In STN, λ-abstractions are leaves.

4. *universal abstractions:* if (v) is a term of type V, then we can form the term $(\Lambda X.v)$ of type $[\Pi X.V]$, so long as the variable X is not free in the type of a free variable of (v). For example, $(\Lambda X.\lambda p^{Int \rightarrow Int \rightarrow X}.p\ 2\ 5)$ has type $[\Pi X.(Int \rightarrow Int \rightarrow X) \rightarrow X]$ which we will see later is the type for pairs of $[Int]$s. In STN, universal abstractions are leaves.

5. *type applications:* if (f) is a term of type $[\Pi X.V]$ and $[U]$ is a type then $(f\ [U])$ is a term of type $[V\{X := [U]\}]$. Type applications are represented in STN by trees in which the left subtree is the tree representation of a term (of abstract type) and the right subtree is a type. The sub-term $((detectFood\ front)\ [Behavior])$ in the term below is such a type application.

$$\begin{array}{l}((detectFood\ front)\ [Behavior] \\ \quad (move\ front) \\ \quad (turn\ ((detectFood\ left)\ [Direction]\ left\ right)))\end{array} \tag{5.4}$$

The STN style tree for this expression is represented in figure 5.2.

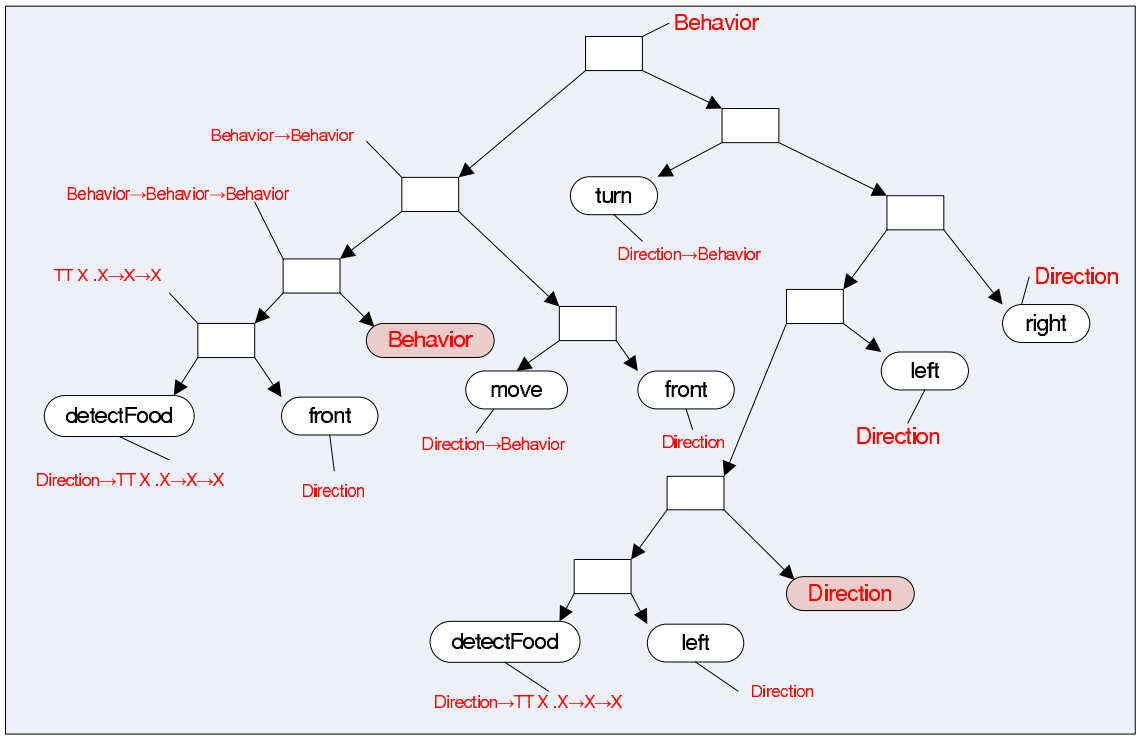

Figure 5.2: The STN parse tree corresponding to term 5.4

5.3.1 α-Conversion and α-Equivalence

Here, the α-conversion operation of the simply-typed system is extended to include types. For example, the type $[\Pi X.X \to X]$ is α-equivalent to the type $[\Pi Y.Y \to Y]$, and the term

$$\lambda x^{[\Pi Y.Y \to Y]}.x\ [\Pi X.X \to X \to X]$$

is α-equivalent to the term

$$\lambda y^{[\Pi Z.Z \to Z]}.y\ [\Pi Y.Y \to Y \to Y]$$

5.3.2 Rewriting, Reduction and Normalization

In its pure form, the system is a rewriting system with only two reduction rules:

1. $((\lambda x^A.v)\ u^A)$ evaluates to $v\{x := u\}$)
2. $((\Lambda X.v)[U])$ evaluates to $(v\{X := [U]\})$

In any λ-calculus, it is said that a term reaches its *normal form* when it can't be reduced any further by the reduction rules of the calculus. The ChurchRosser theorem [13] states that if there are two distinct reductions starting from the same term, then there exists a term that is reachable via a (possibly empty) sequence of reductions from both reducts. As a consequence of the theorem, a term in the λ-calculus has at most one normal form, justifying reference to "the normal form" of a certain term. The ChurchRosser theorem extends to System F [42] and System F is strongly normalizable, meaning that every sequence of reductions starting with a term terminates in a unique normal form.

5.3.3 Normalization and Execution

When we say that a term (a) *normalizes* to a term (b), we mean that (b) is the normal form of (a). Illustrating the first evaluation rule, the term $((\lambda x^{Int}.plus\ 5\ x)\ 4)$ normalizes to $(plus\ 5\ 4)$ and so does the term $((\lambda x^{Int\to Int\to Int}.f\ 5\ 4)\ plus)$. The problem is that the term $(plus\ 5\ 4)$ does not normalize to the term (9). In order to get the value 9 from the term $(plus\ 5\ 4)$, it is necessary to step outside the normalization system, into some external execution procedure that may return a value or another term that might kick-start the normalization process again.

A *pure abstraction*, like the term of equation 5.5, expresses behavior independently of type consideration.

$$(\Lambda Y.\Lambda X.\lambda x^{X}\lambda f^{X\to Y}.f\ x) \tag{5.5}$$

There are no dependencies between what such a term computes and the types of its arguments. In a sense, term 5.5 expresses *only* general program behavior. When terms are pure abstractions, execution is just another name for normalization, but GP specific problems require that we extend the System F by adding term and type constants. The use of System F in a GP context also requires that a method to execute programs (execution here is used in the sense of the $\chi_{(Ph)}$ function of definition 2.1) be introduced.

Next, we provide an example of the interplay between normalization and execution. Starting with the term:

$$\begin{array}{c}(\lambda x^{Int}.(gt\ (plus\ 6\ 4)\ time\ [Int\to Int]\ (plus\ 2)\ (minus\ 3))\ x)\\ (gt\ time\ 10\ [Int]\ time\ 10)\end{array} \tag{5.6}$$

The term 5.6 reuses the definitions introduced in 4.3.1, in particular, the function gt is typed as $[Int \to Int \to (\Pi X.X \to X \to X)]$. Step (1) happens outside the rewriting system. The sub-term $(gt\ time\ 10)$ is evaluated first by the execution sub-system, which must return an object of type $[\Pi X.X \to X \to X]$, for example $(\Lambda X.\lambda x^{X}.\lambda y^{X}.x)$, yielding:

$1 : (\lambda x^{Int}.(gt\ (plus\ 6\ 4)\ time\ [Int\to Int]\ (plus\ 2)\ (minus\ 3))\ x)$
$((\Lambda X.\lambda x^{X}.\lambda y^{X}.x)[Int]\ time\ 10)$

Step (2) stays within the rewriting system:
$2 : (\lambda x^{Int}.(gt\ (plus\ 6\ 4)\ time\ [Int\to Int]\ (plus\ 2)\ (minus\ 3))\ x)$
$((\lambda x^{[Int]}.\lambda y^{[Int]}.x)\ time\ 10)$

Step (3) is a combination of two reduction steps. It also remains within the rewriting system:
$3 : (gt\ (plus\ 6\ 4)\ time\ [Int\to Int]\ (plus\ 2)\ (minus\ 3))\ time$

In Step (4) (this is arbitrary, but it has to happen at some point) we step outside the rewriting system, into the execution system. The execution system will take the sub-term $(plus\ 6\ 4)$ and replace it with the sub-term (10), resulting in:
$4 : (gt\ 10\ time\ [Int\to Int]\ (plus\ 2)\ (minus\ 3))\ time$

Step (5) executes the sub-term $(gt\ 10\ time)$, returning a sub-term to replace it with. Again, the result of this execution is a term of type $[\Pi X.X \to X \to X]$, for example $(\Lambda X.\lambda x^X.\lambda y^X.x)$, yielding:
$5 : ((\Lambda X.\lambda x^X.\lambda y^X.x)\ [Int \to Int]\ (plus\ 2)\ (minus\ 3))\ time$

Step (6) stays within the rewriter, and yields:
$6 : (\lambda x^{[Int \to Int]}.\lambda y^{[Int \to Int]}.x)\ (plus\ 2)\ (minus\ 3))\ time$

Step (7), also a combination of two reduction steps, stays within the rewriter, and yields:
$7 : plus\ 2\ time$

Step (8) is to exit the rewriting rule system and never return. It is completely within the execution function's domain, which has information about the $time$ variable, and is able to evaluate a $plus$ operation.

As another example, by defining the name $numpair$ as an equivalent for the term:
$(\lambda x^{Int}.\lambda y^{Int}.\Lambda X.\lambda p^{Int \to Int \to X}.p\ x\ y)$, we can express a pair of numbers (a, b) by the term $(numpair\ a\ b)$. The term $(numpair\ 4\ 5\ [Int]\ (\lambda x^{Int}.\lambda y^{Int}.x))$ will use the second rewriting rule to eventually normalize to the term (4) (in 6 steps) while the term $(numpair\ 4\ 5\ [Int]\ (\lambda x^{Int}.\lambda y^{Int}.y))$ will normalize to the term (5). These two terms express the projection operations of a pair.

5.4 Abstract Data Types

The novelty of this work is based on this almost magical fact: Type abstraction can be used to form types that describe structures. It turns out that not only do types describe structures, they almost mechanically produce the tools with which the structures they describe can be used. A lot of the basic algorithms on data structures we may think to have been "invented" seem to turn out to exist as an inherent part of the structures. For examples, it seems that simply describing the structure of a tree in system F is enough to derive the algorithm for recursive tree traversal. It seems that the implementations for the operations for $fold$, map and $iter$ can be almost mechanically derived once the structure for a list has been described by a type. In this section, several type and object definitions are presented as examples. With these examples, we show how types and functions may be defined as pure abstractions, describing *structure*. This implies that any genetic program whose representation scheme is based on System F would have the capacity to express the examples below, no matter what the set of functions that are defined for the genetic program is. There is another way to say this:

> **Any GP system based on System F that has the capacity to name and to reuse terms and types also has the capacity to represent structures such as booleans, numbers, pairs, lists, trees, tuples and all the usual operations on those structures. In addition, such a system can represent recursive functions. The representation is encoded in the structure of the tree and is independent of the primitive terms and types that are defined for the GP system.**

5.4.1 Finding the Representations

System F comes with a general scheme for the representation of free structures. To generate objects of type $[Ty]$, where $[Ty]$ represents a structure, a series of constructors is needed. These are functions that take no, one, or several arguments and use them to produce objects of type $[Ty]$.

Definition 5.4.1 (Constructor) Given a type $[A]$, a constructor for $[A]$ is defined as:

1. An object of type $[A]$ (called a primary constructor)
2. An object of type $[B \to C]$ where $[C]$ is the type of a constructor for an object of type $[A]$

Constructors can be recursive. For example the $cons$ constructor for lists of objects of type $[A]$ creates a list from two arguments: a list of elements of a given type, and an element of type $[A]$. This gives it the type:

$$[A \to List \to List]$$

The type $[List \to A \to List]$ also works.

5.4.2 Format for the Types

The representation of data types follows a specific pattern, outlined in [42]. An algorithm to identify such types is presented in section 6.2.2 (Program 6.2.3). Data types are in the form:

$$\Pi X.S_1 \to \ldots \to S_n \to X$$

with each S_i the type of a constructor for X, and therefore in the form:

$$T_1^i \to \ldots \to T_{k_i}^i \to X$$

As a result of this structure, for any type $[Ty]$, each $S_i\{X := [Ty]\}$ is the type of a constructor for an object of type $[Ty]$. This implies that each structure requires $f_1, \ldots, f_n$ constructor definitions. For example, if $[Nat]$ is the type of natural numbers, there must be two constructors, the first one, typed as $[Nat \to Nat]$ (a constructor that takes an integer object and returns its successor, itself an integer object) and the other with type $[Nat]$ coding 0 (with no argument). Similarly, if $[Boolean]$ is the boolean type, it will require 2 constructors, neither of which take any arguments: one that constructs the object $false$ and the other to produce $true$. We haven't found an algorithm that automatically derives constructors from appropriately formed type structures, but in many cases the derivation is trivial (for example, for the terms for $true$ and $false$ for the boolean type $[\Pi X.X \to X \to X]$).

5.4.3 Representing Structures Built from Constants

The simplest data structure is the one that represents a set of distinct constants. For example, the structure representing playing cards is a set of 52 distinct constants and it is isomorphic to any other set of 52 constants. The type Boolean is a name for a set of two constants and it is isomorphic to any other set of two constants. Objects built following this structure pattern are usually used to *choose* from a finite set of possibilities. They are objects that always normalize to one of n possible states. For example, a type that represents a structure made from three constants is:

$$[Three\_consts] \stackrel{\text{def}}{=} [\Pi X.X \to X \to X \to X] \tag{5.7}$$

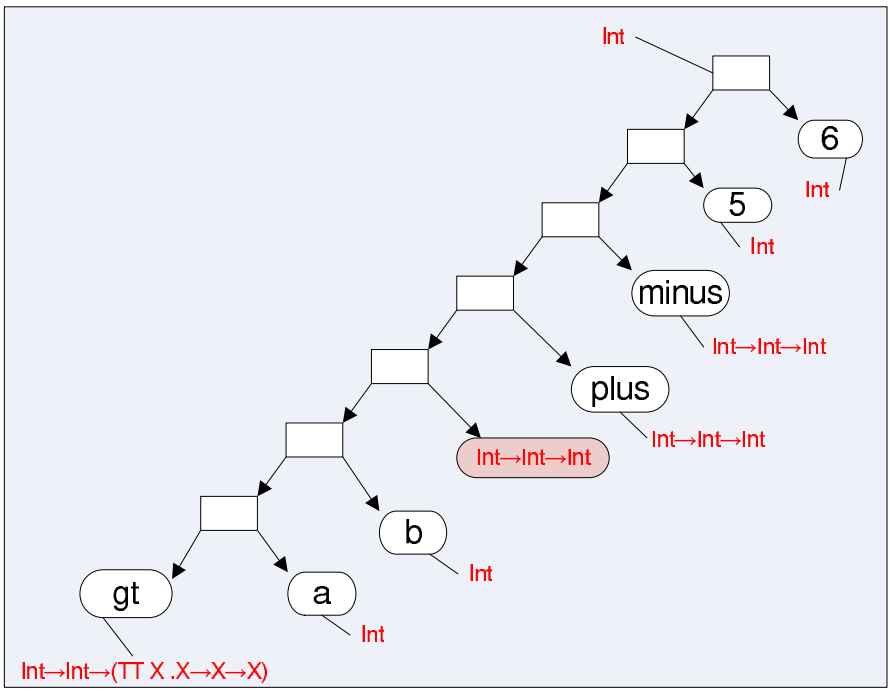

Figure 5.3: A parse tree of the System F term $(gt\ a\ b\ [Int \to Int \to Int]\ plus\ minus\ 5\ 6)$

and an object of this type, for example $traffic\_light\_state$ would typically be used in a context such as:

$$(traffic\_light\_state\ [MotionType]\ stop\ slow\ forward) \tag{5.8}$$

which is an object of type $[MotionType]$. The assumption is, of course, that $traffic\_light\_state$ would evaluate to one of the three constructors derivable for the $[Three\_consts]$ type (conveniently named for this specific purpose):

$$\begin{aligned} red &\stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.\lambda x_3^X.x_1^X \\ yellow &\stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.\lambda x_3^X.x_2^X \\ green &\stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.\lambda x_3^X.x_3^X \end{aligned}$$

As another example, a structure made from four constants is expressed by the type:

$$[Four\_consts] \stackrel{\text{def}}{=} [\Pi X.X \to X \to X \to X \to X] \tag{5.9}$$

and objects of this type must normalize to one of four possible states.

5.4.3.1 Booleans and Conditional:

The type $[Boolean]$ describes a structure that is built from two constants. The type is defined as:

$$[Boolean] \stackrel{\text{def}}{=} [\Pi X.X \to X \to X] \tag{5.10}$$

The terms $true$ and $false$ are defined as pure abstractions and there are only two possibilities. Beginning with

$$S_1 \stackrel{\text{def}}{=} X \qquad\qquad S_2 \stackrel{\text{def}}{=} X \tag{5.11}$$

Implying that there are two constructors, of type $[S_1\{X := Boolean\}]$ and $[S_2\{X := Boolean\}]$ (two constants):

$$true \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.x_1^X \qquad\qquad false \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.x_2^X \tag{5.12}$$

We might also switch the expressions for true and false, and everything would still work just the same (with different expressions). Having an abstract definition of the boolean type means a GP system does not need any additional syntax to express branching. Boolean objects become their own decision operators. For example, if our terminal set includes a function "greater than", typed as: $gt : [Int \rightarrow Int \rightarrow Boolean]$, then the expression $(gt\ a\ b\ [Int]\ 1\ 2)$ expresses the program:"if (a) evaluates to something greater than the evaluation of (b), then this program evaluates to (1), otherwise it evaluates to (2)". This works because the expression $(true\ [Int]\ 1\ 2)$ evaluates to the term (1) while the expression $(false\ [Int]\ 1\ 2)$ evaluates to the term (2). Similarly,

$$gt\ a\ b\ [Int \rightarrow Int \rightarrow Int]\ plus\ minus\ 5\ 6 \tag{5.13}$$

expresses the sentence: "if (a) evaluates to something greater than (b) then evaluate to $(plus\ 5\ 6)$, otherwise evaluate to $(minus\ 5\ 6)$", assuming of course the pre-definition of the terminal $minus$, of type $[Int \rightarrow Int \rightarrow Int]$. The parse tree for this expression is depicted in figure 5.3. This is just a special case of the n constant structure where the derived conditional behaves like a case statement rather than like an if_then_else function.

5.4.4 Representing Tuples

The most common data structure is a tuple or conjunction. The structure groups objects of different types into one unit structure. The special case type $[Pair_{AB}]$, for a pair of two objects of respectively of type [A] and [B] is definable as:

$$[Pair_{AB}] \stackrel{\text{def}}{=} [\Pi X.(A \rightarrow B \rightarrow X) \rightarrow X] \tag{5.14}$$

This implies that there is only one constructor, of type $[A \rightarrow B \rightarrow Pair_{AB}]$:

$$pair \stackrel{\text{def}}{=} \lambda a^A.\lambda b^B.\Lambda X.\lambda x_1^{A\rightarrow B\rightarrow X}.x_1\ a\ b \tag{5.15}$$

so that $(pair\ a^A\ b^B)$ is an object of type $[Pair_{AB}]$. The type can be made more general, for example $[\Pi A.\Pi X.(A \rightarrow B \rightarrow X) \rightarrow X]$ describes a data structure built from the conjunction of two objects, the second of which is an object of type [B] and the first is left undefined. To access the members of the structure, projection terms are needed. For example,

$$left \stackrel{\text{def}}{=} \lambda p^{Pair_{AB}}.p\ [A]\ (\lambda a^A.\lambda b^B.a) \tag{5.16}$$

so that $(left\ (pair\ a\ b))$ normalizes to a. This operation can also be made more general by abstracting one or both types. The structure corresponding to the type $[\Pi X.(A \rightarrow B \rightarrow C \rightarrow X) \rightarrow X]$ is of course a

tuple with three elements, respectively of type [A], [B] and [C]. For example, $[TrInt]$, a type that describes a structure that groups three integers is definable as:

$$[TrInt] \stackrel{\text{def}}{=} [\Pi X.(Int \to Int \to Int \to X) \to X] \tag{5.17}$$

$[TrInt]$ is in the usual format and the constructor scheme can be applied to yield a constructor:

$$tr \stackrel{\text{def}}{=} \lambda i^{Int} \lambda j^{Int} \lambda k^{Int} \Lambda X \lambda x^{Int \to Int \to Int \to X}.x\ i\ j\ k \tag{5.18}$$

so that an object $(tr\ 5\ time\ 1)$ will normalize to the term:

$$\Lambda X.\lambda x^{(Int \to Int \to Int \to X) \to X}.x\ 5\ time\ 1 \tag{5.19}$$

Operations on the structure are almost immediately derivable. For example to obtain a $[TrInt]$ from $t : [TrInt]$ with its last variable incremented, all that's needed is to suffix t with the appropriate handler:

$$t\ [TrInt]\ (\lambda x^{Int} \lambda y^{Int}.\lambda z^{Int}.tr\ x\ y\ (plus\ z\ 1) \tag{5.20}$$

whereas to obtain the sum of all the elements:

$$t\ [Int]\ (\lambda x^{Int} \lambda y^{Int}.\lambda z^{Int}.plus\ x\ (plus\ y\ z) \tag{5.21}$$

5.4.5 Representing Unions

Unions are used to model containers that contain one of several possible objects. For example, the type $[Polygon]$, a wrapper for an object that can be either of type $[Square]$, $[Rectangle]$ or $[Triangle]$ is definable as:

$$[Polygon] \stackrel{\text{def}}{=} [\Pi X.(Square \to X) \to (Rectangle \to X) \to (Triangle \to X) \to X] \tag{5.22}$$

There is no way to tell what is inside the union from the outside. So the only way to make use of it, is to pass handlers for all possible cases. For example, the constructor that constructs a union of two objects of type [A] and [B] from an object of type [A] is definable as:

$$union_a \stackrel{\text{def}}{=} \lambda a^A.\Lambda X.\lambda x_1^{A \to X}.\lambda x_2^{B \to X}.x_1\ a \tag{5.23}$$

To work with $(union_a\ a^A)$, an object of type $[\Pi X.(A \to X) \to (B \to X) \to X]$, one must have a handler for both possible contents. For example, $(union_a\ a^A\ [C]\ f_1\ f_2)$ is an object of type $[C]$, with handlers f_1 of type $[A \to C]$ and f_2 of type $[B \to C]$. Unions are also useful to describe containers that may be empty. For example, a $[Box]$ may be empty or may contain an object of type $[Toy]$, which produces the descriptions:

$$[Box] \stackrel{\text{def}}{=} [\Pi X.X \to (Toy \to X) \to X] \tag{5.24}$$

So if we define $play : [Toy \to Behavior]$ and $loiter : [Behavior]$ then expressing the sentence "if there is a toy in the box then play with it, otherwise loiter" is done with:

$$box\ [Behavior]\ loiter\ play \tag{5.25}$$

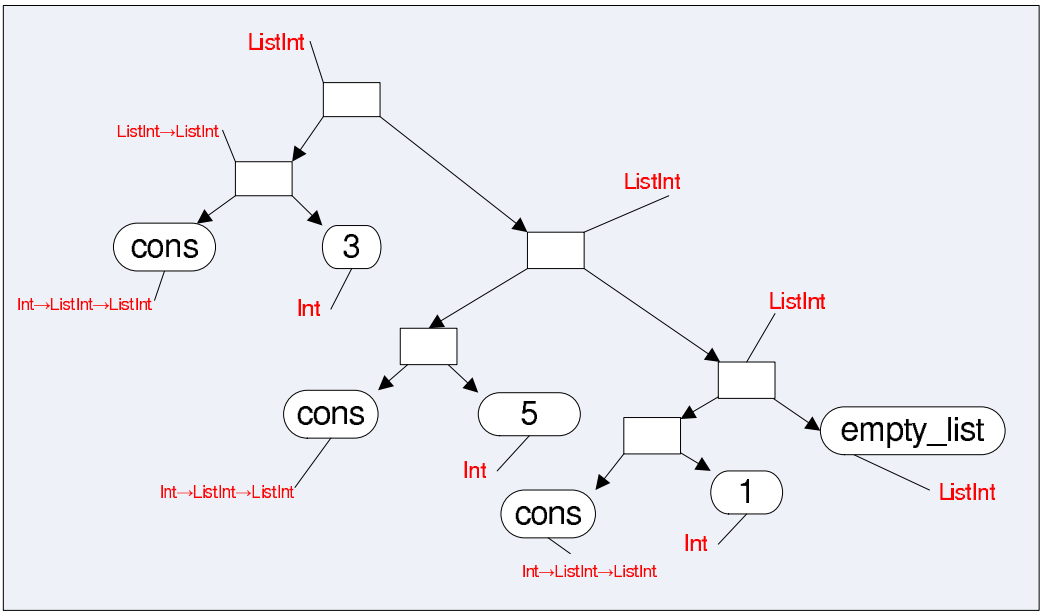

Figure 5.4: Parse tree for list of three objects of type [Int]

which is an object of type [Behavior] (box is an instance of [Box]). Unions can also model inheritance relationships, so if [Toy] is defined as:

$$[Toy] \stackrel{\text{def}}{=} [\Pi X.(Puzzle \to X) \to (Ball \to X) \to X] \tag{5.26}$$

then $play : [Toy \to Behavior]$ can be seen as a method of an object of type $[Toy]$, overloaded on $[Toy]$'s sub-types $[Puzzle]$ and $[Ball]$ and the overloading can be defined within System F:

$$play \stackrel{\text{def}}{=} \lambda t^{Toy}.t\ [Behavior]\ solve\ kick \tag{5.27}$$

with $solve : [Puzzle \to Behavior]$ and $kick : [Ball \to Behavior]$.

5.4.6 Representing Lists and Some Derived Recursive Operations

The type for lists also follows the structure of a list, an object that is either built from an empty list or from the consing of an object and a list of objects of the same type. The most general type for lists is:

$$[\Pi Y.\Pi X.X \to (Y \to X \to X) \to X] \tag{5.28}$$

The type $[List_U]$ of a list of objects of type $[U]$ is definable as:

$$[List_U] \stackrel{\text{def}}{=} [\Pi X.X \to (U \to X \to X) \to X] \tag{5.29}$$

So for example, an object of type $[List_{Int}]$ with elements 3, 5 and 1 looks like the term below when normalized:

$$lst_{[Int]} \stackrel{\text{def}}{=} ((\Lambda X.\lambda x^X.\lambda y^{Int \to X \to X}.y\ 3\ (y\ 5\ (y\ 1\ x))) \tag{5.30}$$

To obtain the constructors for the type $List_U$ we starts by identifying the two S_i:

$$S_1 \stackrel{\text{def}}{=} X \qquad\qquad S_2 \stackrel{\text{def}}{=} U \to X \to X \tag{5.31}$$

From which we can derive the following two constructors:

$$\begin{aligned} &empty\_list \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^{U\to X\to X}.x_1 \\ &cons \stackrel{\text{def}}{=} \lambda p_1^U.\lambda p_2^{List_U}.\Lambda X.\lambda x_1^X.\lambda x_2^{U\to X\to X}.x_2\ p_1\ (p_2\ X\ x_1\ x_2) \end{aligned} \tag{5.32}$$

The second constructor of the list data type illustrates how recursion happens. The polymorphic argument p_2 is first applied to the type variable X, which gives it the right type to "fit" inside the body of the constructor and makes it accept the arguments x_1 and x_2. Figure 5.4 is the parse tree expression of

$$(cons\ 3\ (cons\ 5\ (cons\ 1\ empty\_list))) \tag{5.33}$$

which is an object of type $List_U$ in which objects of type [Int] are used for the type [U]. It expresses the list of three numbers $(3 :: 5 :: 1 :: empty\_list)$, corresponding to the term 5.30. The normalized form of a list $(u_1, u_2, \ldots, u_n)$ of elements of type $[U]$ is encoded as:

$$\Lambda X.\lambda x^X.\lambda y^{\mathrm{U}\to\ X\ \to\ X}.y\ u_1(y\ u_2(\ldots(y\ u_n\ x))\ldots) \tag{5.34}$$

Recursive Programming with Lists: Creating a recursive procedure essentially requires defining a "base case", and then defining rules to break down more complex cases into the base case. The key to a recursive procedure is that with each recursive call, the problem domain must be reduced in such a way that eventually the base case is arrived at. So being able to describe lists immediately provides a convenient, guaranteed to terminate recursion scheme. For example,

$$(lst_{[Int]}\ [Int]\ 0\ plus) \tag{5.35}$$

The parse tree of 5.35 is depicted in figure 5.5. 5.35 expresses the recursive addition of the elements of $lst_{[Int]}$. It normalizes to the term $(plus\ 3\ (plus\ 5\ (plus\ 4\ 0)))$ in three steps, demonstrating how simple it is to piggy-back onto the list's recursive structure to express the recursive behavior known as "folding". Similarly, the term:

$$((cons\ a\ (cons\ b\ (cons\ c\ empty\_list)))\ [Int]\ 1\ mult) \tag{5.36}$$

will normalize to $(mult\ a\ (mult\ b\ (mult\ c\ 1)))$ and expresses the recursive multiplication of the elements of the list.

Expressing the *mapping* of a list to another list, is just as simple. We simply need to redefine the $cons$ operation to something else and pass it as an argument to the list. It is the internal structure of the list that does all the work of modifying the elements and putting them back together. For example,

$$lst_{[Int]}\ [List_{Int}]\ empty\_list\ (\lambda x^{Int}.\lambda l^{List_{Int}}.cons\ (plus\ 1\ x)\ l)) \tag{5.37}$$

expresses the program that takes our example list and maps it to a list in which every element has been incremented. Recursion happens so naturally because the list is a function that takes a type as its argument, and it is the type argument that makes the list compatible with the arguments that come after.

5.4.7 Representing Trees

If we describe a tree structure as something that is either a node containing an object of type $[A]$ or a node joining a left and a right tree, then the type that describes the structure is:

$$[Tree_1] \stackrel{\text{def}}{=} [\Pi X.(A \to X) \to (X \to X \to X) \to X] \tag{5.38}$$

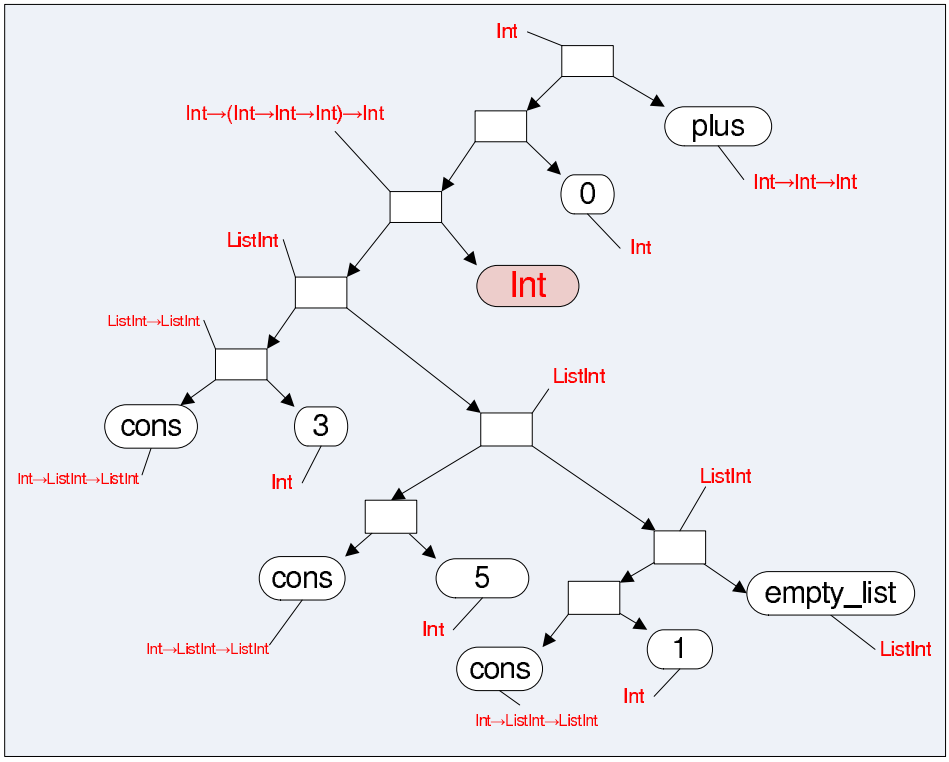

Figure 5.5: Recursive sum on a List

$leaf$ is the constructor that takes an object of type $[A]$ and builds an object of type $[Tree_1]$. It can be trivially obtained with the construction scheme:

$$leaf \stackrel{\text{def}}{=} \lambda a^{A}.\Lambda X.\lambda x_1^{A\to X}.\lambda x_2^{X\to X\to X}.x_1\ a \tag{5.39}$$

The $merge$ constructor builds a tree from two sub-trees and is also obtained mechanically from the type:

$$merge \stackrel{\text{def}}{=} \lambda t_1^{Tree_1}.\lambda t_2^{Tree_1}\Lambda X.\lambda x_1^{A\to X}.\lambda x_2^{X\to X\to X}.x_2\ (t_1\ [X]\ x_1\ x_2)\ (t_2\ [X]\ x_1\ x_2) \tag{5.40}$$

so that $(merge\ (leaf\ a_1)\ (merge\ (leaf\ a_2)\ (leaf\ a_3)))$ represents a tree with three leaves built on this format. If we want to change the representation scheme so that inner nodes contain objects of type $[B]$, then the type representing this structure is:

$$[Tree_2] \stackrel{\text{def}}{=} [\Pi X.(A\to X)\to(B\to X\to X\to X)\to X] \tag{5.41}$$

.

5.4.8 Iteration By Natural Numbers

The representation for natural numbers requires two constructors: a constant (for 0) of type $[Nat]$ and the successor function of type $[Nat\to Nat]$:

$$[Nat] \stackrel{\text{def}}{=} [\Pi X.(X\to X)\to X\to X] \tag{5.42}$$

And there are two S_i:

$$S_1 \stackrel{\text{def}}{=} X \to X \qquad\qquad S_2 \stackrel{\text{def}}{=} X \tag{5.43}$$

So we need two constructors:

$$\begin{array}{l} succ \stackrel{\text{def}}{=} \lambda p_1^{Nat}.\Lambda X.\lambda x_1^{X\to X}\lambda x_2^X.x_1\ (p_1\ X\ x_1\ x_2) \\ zero \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^{X\to X}\lambda x_2^X.x_2 \end{array} \tag{5.44}$$

The $[Nat]$ object n is the function which to any type $[U]$ and function f of type $[U \to U]$ associates the function f^n, i.e. f iterated n times.

Fibonacci Example: We now present a System F program to compute the Fibonacci sequence. To do this, we define the name *fib* as:

$fib \stackrel{\text{def}}{=} \lambda d^{TrInt}.$
$\quad tr\ (plus\ (d\ Int\ (\lambda x^{Int}\lambda y^{Int}\lambda z^{Int}.x))\ 1)$
$\quad\quad (d\ Int\ (\lambda x^{Int}\lambda y^{Int}\lambda z^{Int}.z))$
$\quad\quad (d\ Int\ (\lambda x^{Int}\lambda y^{Int}\lambda z^{Int}.(plus\ \ y\ z)))$

of type $[TrInt \to TrInt]$. fib has an invariant property on objects of type $[TrInt]$: Given j, the $(n-1)^{st}$ term, and k, the n^{th} term in the Fibonacci sequence then $(fib\ (tr\ n\ j\ k))$ evaluates to $(tr\ (+\ n\ 1)\ k\ (+\ j\ k))$. We can now compute the Fibonacci sequence for $n+2$ in many different ways. For example using $[Nat]$s :

$$(te\_n\ [TrInt]\ fib\ (tr\ 2\ 0\ 1)) \tag{5.45}$$

where te_n is the n^{th} numeral of type $[Nat]$. We could also use lists and write:

$$te\_l\ [TrInt]\ (tr\ 2\ 0\ 1)\ (\lambda u^{[Ty]}\lambda t^{TrInt}.fib\ t) \tag{5.46}$$

where te_l is a list of n elements of any type $[Ty]$. There are many other possibilities for the computation.

5.5 CONCLUSION

A recursive function f which is total from $\mathbb{N}$ to $\mathbb{N}$ is called provably total in a system of arithmetic A if A proves the formula which expresses "for all n, the program e, with input n, terminates and returns an integer" for some algorithm e representing f. The precise formulation depends on how programs are formally written in A. System F can represent all provably total functions. However, System F's strong normalization property implies that its programs will always eventually terminate. This in turns implies (by the unsolvability of the halting problem [75]) that there are computable functions that cannot be represented in System F. This is not so bad as it sounds because as [3] puts it, in order to find computable functions that cannot be represented in F, "one has to stand on one's head". In theory, we could do all the programming we would ever need without going outside the pure system. In practice, the genetic programs we experiment with include primitive external functions and terminals defined as symbols and evaluated using a semantic evaluator that gives them meaning outside their System F representation.

In the next chapter, we describe how our work uses System F as the underlying language for the representation of genotypes in GP.

Chapter 6

Abstraction-Based Genetic Programming

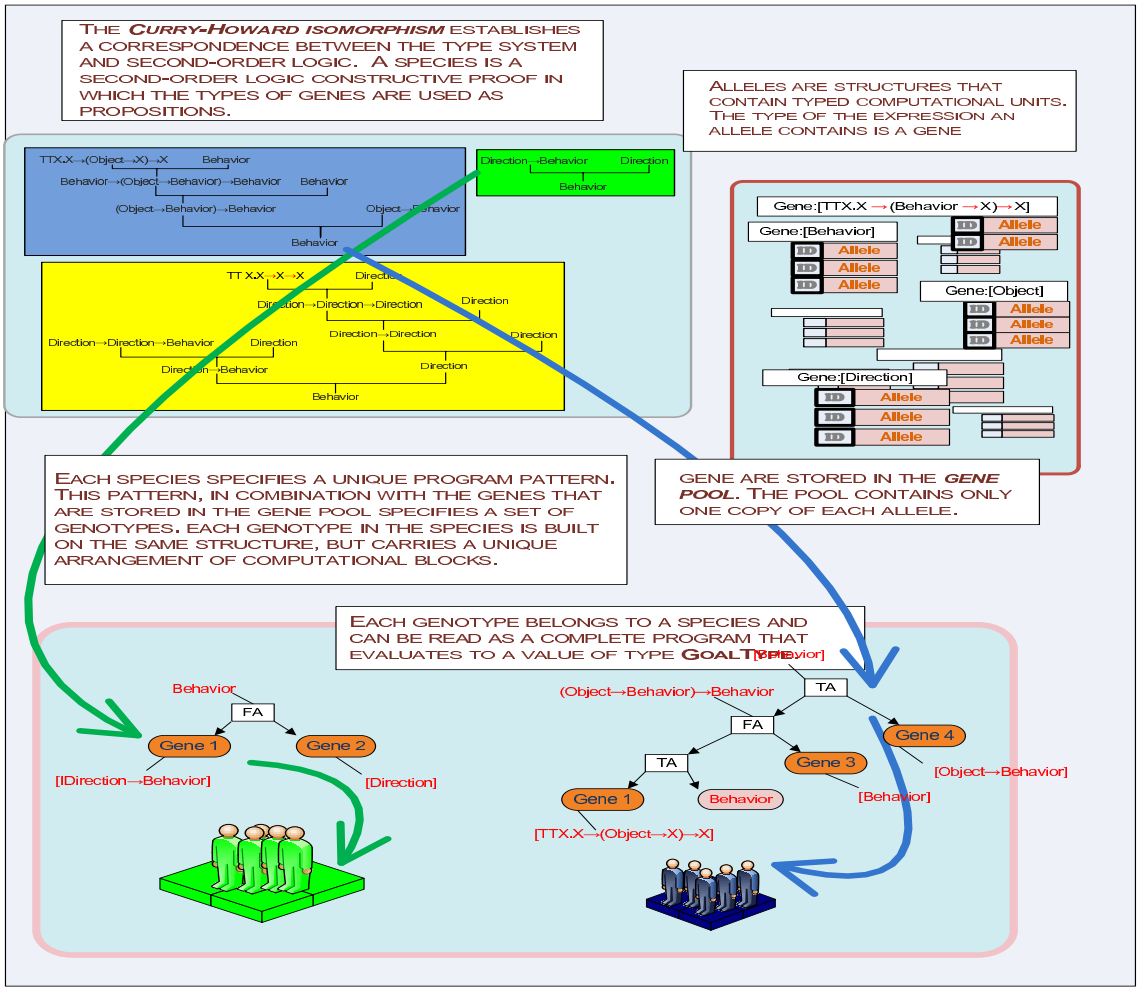

Figure 6.1: Overview of the relationship between species, alleles and genotypes

Figure 6.1 provides a diagram overview of Abstraction-Based Genetic Programming (ABGP). In ABGP, genotypes are aggregates of anonymous computational building blocks called *alleles* and of types. Each allele contains a typed System F term. The type of the term enclosed by the allele specifies the location of a genotype's structure where it may be used as a gene. The type is called *gene* and each gene has several alleles. Genotypes combine alleles with other alleles using two combination operations, corresponding to the operations of functional and type application introduced in 5.3. Alleles are compact atomic computational blocks. The type partially specifies what the program (abstractly) does. It is also an instruction as to where its alleles may be inserted into a genotype. Alleles are closed and can't be opened. They can only be used or not. The type of the set of alleles completely determines all the possible pluggings it allows without crashing. One can always substitute the allele of a given gene with another allele of the same gene in a genotype to produce a different, but valid genotype.

Each genotype follows a pattern in the way in which it combines genes. This pattern is specified by the *species* to which the genotype belongs. A species is a proof and a genotype belongs to a species when the typing structure of its gene arrangement is Curry-Howard isomorphic to it.

6.1 NOTATION AND PROGRAMMING SYNTAX

6.1.1 Presentation of Algorithms and Pseudo-Code

Program 6.1.1 The $match$ syntactic structure

```
match a with
   | b --> block1
   | c when predicate --> block2
   | _ --> block3
```

When we present the algorithms, we use the "match" syntax (reflected from our Objective Caml implementation). For example, in program 6.1.1, $block1$ is to be executed if a matches b (on a structural level), while $block2$ should be executed when a matches c and $predicate$ evaluates to true. The last block, $block3$ is executed if none of the cases above it have been matched (_ stands for anything). Order matters, so while there may be several matches to one argument, it is the first match that is executed.

We also use some common data structures when we present pseudo-code. We write $a : (B, C)\ Hashtbl$ to indicate that a is a hash table in which objects of type B index objects of type C. We write $A\ list$ to denote the type of a list of objects of type A, and $(A * B)$ to denote the type obtained from a pair of objects of type A and B respectively.

6.1.2 Output/Input from Implementation as Examples

We often use text input or output directly from the implementation that we constructed as examples. When we do this, we use an ascii text format for representing expressions that often contain greek letters and other symbols. Our implementation's lexer reads:

1. 'Lam' as Λ
2. 'lam' as λ
3. 'TT' as Π
4. '->' as $\rightarrow$

6.2 PRIMITIVES

To represent System F terms and types, we use the implementation of the de Bruijn canonical representation of variables and expressions [8] described in [63]. It has the advantage of making α-equivalence the same as syntactic equality. In this representation scheme, named variables are replaced by natural numbers, each index is a number that represents an occurrence of a variable in an expression, and denotes the number of binders that are in scope between that occurrence and its corresponding binder. Variable occurrences point directly to their binders, rather than being referenced by name. To use this representation scheme, the reading of an expression needs to be done in conjunction with a context.

6.2.1 Contexts

context 6.2.1 $ListStr$, a context for evolving list operations

```
/*****************************
Context ListStr
for strings and list of strings programs
*****************************/

TyList = TT X . X -> (String->X->X)->X;
TyBool = TT X . X -> X ->X;

true = Lam X . lam x : X . lam y : X . x;
false = Lam X . lam x : X . lam y : X . y;

empty_list  =   Lam X . lam x : X . lam y : String->X->X . x;
cons        =   lam elt : String .
                lam lst : TyList .
                Lam X . lam x : X . lam y : String->X->X . y elt (lst [X] x y);

empty_string = "";

length : String->Int;
concat : String->String->String;
compstr : String -> String -> TyBool;
substr : String -> Int -> Int -> String;
```

A *context*, such as the one displayed in 6.2.1 contains the necessary information needed to read a term. The typing relationships of System F cannot be represented in Backus-Naur form, thus, the use of System F as a representation scheme requires the use of a context which is simply a list of indexed names associated with relevant information. The formal definition of a context is below:

Definition 6.2.1 (Context)

```
Context ::= (<String> * <Binding>) list
```

In which the object of type binding contains the relevant information needed to use the name appropriately. Each term is associated with a context, and when a term contains abstractions, the context associated with the inner sub-terms (inner contexts) of the term are extensions of the outer context that include additional elements corresponding to the variables (types or typed term variables) that exist within the scope of the inner term. For example, in the term of 6.1,

$$\underbrace{\lambda x^{Int}.\,\lambda y^{Int}.\quad \overbrace{\overbrace{plus\ x\ y}^{\text{context 3 (innermost)}}}^{\text{context 2}}}_{\text{context 1 (outermost)}} \tag{6.1}$$

a suitable outer context to represent this term would need to contain (at a minimum) the symbol $plus$ and its type, $[Int \rightarrow Int \rightarrow Int]$. Context 2, the context for the $(\lambda y^{Int}.plus\ x\ y)$ part of term 6.1 is the outermost context extended with the variable $x : Int$. Context 3, the context associated with the $(plus\ x\ y)$ sub-term is context 2 extended with the variable $y : Int$. We pre-included the types $Unit$, Int, $String$ and $Float$ into the system. "Pre-including" in this case means that these types and their objects are part of the underlying implementation rather than of the representation scheme. The definitions that are normally provided to a GP system through its terminal set are also provided via the context. The context file is translated to a list of names and their associated definition. For example, in our implementation, context 6.2.1 is stored into something similar to the following list (with the context indexes on the left):

```
index   Name           Binding
0       substr:         Term constant of type String-> Int-> Int-> String
1       compstr:        Term constant of type String-> String-> TyBool
2       concat:         Term constant of type String-> String-> String
3       length:         Term constant of type String-> Int
4       empty_string:   Name For Composed Term ""
5       cons:           Name For Composed Term lam d:String.lam d':TyList.
                        ...Lam K.lam s:K.lam f:String-> K-> K.f d (d' [K] s f)
6       empty_list:     Name For Composed Term Lam V.lam j:V.lam h:String-> V-> V.j
7       false:          Name For Composed Term Lam J.lam v:J.lam y:J.y
8       true:           Name For Composed Term Lam L.lam p:L.lam r:L.p
9       TyBool:         Name For Composed Type TT A. A-> A-> A
10      TyList:         Name For Composed Type TT X. X-> (String-> X-> X)-> X
```

The names defined in a context may refer to objects that exist outside System F or to constructs that exist within it. Unlike other GP representation systems, in which primitives are only names for the functions and terminals used to construct genotypes, in an Abstraction-Based Genetic Programming System, primitives may also be used to encode known information about the relationships that exist within the solution space. The information relevant to the each primitive is stored in the binding object associated with its name. Primitives may be:

1. *Atomic types*, used as names for sets of objects. For example, the types $[Food]$, $[Obstacle]$ or $[Behavior]$.

2. *Names for Composite Types*, used to describe concepts related to the problem. For example, defining the type $[Object]$ as:

$$[\Pi X.(Food \to X) \to (Obstacle \to X) \to (Pheromone \to X) \to X] \tag{6.2}$$

   indicates that an object of type $[Object]$ may be either an object of type $[Food]$, $[Obstacle]$ or $[Pheromone]$. Some types may be included as either atomic or composite. For example, the type $[Boolean]$ might be included as a name for the type $[\Pi X.X \to X \to X]$ or as an atomic type. The system's user may also choose to include some composite types as a mechanism to simplify expressions. For example, the type $[Int \to Int \to Int]$ might be renamed as the shorter $[BiOp]$.

3. *Typed free variables* are used to define or name concepts with an execution or evaluation procedure outside System F normalization. Free variables are included in $\Im_{ty}$ in the form $(\alpha : \Gamma)$, with Γ the type of the variable α. For example, $(6 : [Int])$, $(plus : [BiOp])$, $(move : [Direction \to Behavior])$ or $(north : [Direction])$

4. *Names for System F terms*, the types of which need not be included as they can be inferred by the system. For example, in table 6.2.2, $true$ is included as a name for the term $(\Lambda X.\lambda x^X.\lambda y^X.x)$ and $detectFood$ as a name for the term

$$\lambda d^{Direction}.(cell\ d)\ [Boolean]$$
$$false$$
$$(\lambda o^{Object}.o\ [Boolean]\ (\lambda f^{Food}.true)\ (\lambda o^{Obstacle}.false)\ (\lambda p^{Pheromone}.false)))$$

   making $detectFood$ an object of type $[Direction \to Boolean]$ with $cell$ defined as the free variable

$$(cell : [Direction \to (\Pi X.X \to (Object \to X) \to X)])$$

   and $[Object]$ defined as in 6.2. The type $[\Pi X.X \to (Object \to X) \to X]$ represents a structure that may contain nothing or an object of type $[Object]$.

Tables 6.2.2 and 6.2.3 contain two sample context files (with expressions represented using mathematical notation).

6.2.2 Representation of Types

Types are built using one of 7 constructors. The exact definition for the representation scheme is:

Definition 6.2.2 ($< Type >$)

```
<Type>   ::=   | TyVar <Int>
               | TyArr <Type> <Type>
               | TyAll <Type>
               | TyString
               | TyFloat
               | TyInt
               | TyUnit
```

The last 4 constructors are the types that we've described as being "built in" the system. The first 3 constructors are used to build composite types. So that, for example, the representation of the $[TyBool]$ constant type,

context 6.2.2 An example context for a sample side-effect ABGP system

Type	
$[Behavior]$	User Defined Atomic Type (UDAT)
$[Obstacle]$	User Defined Atomic Type (UDAT)
$[Food]$	User Defined Atomic Type (UDAT)
$[Pheromone]$	User Defined Atomic Type (UDAT)
$[Direction]$	User Defined Atomic Type (UDAT)
Type Name	**Type**
$[Object]$	$[\Pi X.(Food \to X) \to (Obstacle \to X) \to (Pheromone \to X) \to X]$
$[Boolean]$	$[\Pi X.X \to X \to X]$
Functions	**Type**
$cell$	$[Direction \to (\Pi X.X \to (Object \to X) \to X)]$
$dropPheromone$	$[Behavior]$
$move$	$[Direction \to Behavior]$
$turn$	$[Direction \to Behavior]$
$pickUpFood$	$[Food \to Behavior]$
Term Name	**Term**
$true$	$\Lambda X.\lambda x^X.\lambda y^Y.x$
$false$	$\Lambda X.\lambda x^X.\lambda y^Y.y$
$detectFood$	$(\lambda d^{Direction}.(cell\ d)\ [Boolean]\ false\ (\lambda o^{Object}.o\ [Boolean]\ (\lambda f^{Food}.true)\ (\lambda o^{Obstacle}.false)\ (\lambda p^{Pheromone}.false)))$
$detectPheromone$	$(\lambda d^{Direction}.(cell\ d)\ [Boolean]\ false\ (\lambda o^{Object}.o\ [Boolean]\ (\lambda f^{Food}.false)\ (\lambda o^{Obstacle}.false)\ (\lambda p^{Pheromone}.true)))$
Terminals	**Type**
$front$	$[Direction]$
$left$	$[Direction]$
$right$	$[Direction]$

when read in conjunction with context 6.2.1 is $(TyVar\ 9)$. However, when read with the same context, the type $[\Pi X.(X \to TyBool \to X) \to X]$ is represented by:

```
TyAll(
  TyArr
    (TyArr (TyVar 0) (TyArr (TyVar 10) (TyVar 0)))
    (TyVar 0))
```

Within the body of the $TyAll$ type constructor, the constant $(TyVar\ 0)$ represents the closest bound variable of the expression and all the other elements of the context have had their index incremented by one. This is the reason for which the representation of the $TyBool$ constant type is $(TyVar\ 10)$ inside the abstraction (as opposed to $(TyVar\ 9)$ outside of it). Compare with the representation of the type

$$[TyBool \to (\Pi X.(X \to TyBool \to X) \to X)] \tag{6.3}$$

context 6.2.3 A context for a sample value ABGP system

Index	Function Name	Type
0	gt	$[TyInt \rightarrow TyInt \rightarrow (\Pi X.X \rightarrow X \rightarrow X)]$
1	$plus$	$[TyInt \rightarrow TyInt \rightarrow TyInt]$
2	div	$[TyInt \rightarrow TyInt \rightarrow TyInt]$
3	$minus$	$[TyInt \rightarrow TyInt \rightarrow TyInt]$
4	$times$	$[TyInt \rightarrow TyInt \rightarrow TyInt]$
Index	**Terminal Name**	**Type**
5	$time$	$[TyInt]$
6	$zero$	$[TyInt]$
7	one	$[TyInt]$

which is represented within the ListStr context by the expression:

```
TyArr
  (TyVar 9)
  (TyAll
    (TyArr
       (TyArr (TyVar 0) (TyArr (TyVar 10) (TyVar 0)))
       (TyVar 0)))
```

The context of an expression may be the result of several shifts. For example, the type:

$$[\Pi Y.Y \rightarrow (\Pi X.X \rightarrow Y) \rightarrow Y] \tag{6.4}$$

is represented as:

```
TyAll(
  TyArr
    (TyVar 0)
    (TyArr
        (TyAll (TyArr (TyVar 0) (TyVar 1)))
        (TyVar 0)))
```

Here, the first bound variable (Y) appears as $(TyVar\ 1)$ in the innermost context, while within the innermost context, $(TyVar\ 0)$ refers to the X bound variable. Note that the representation for this term is invariant under any context. This is because the type is a pure abstraction. Note also that it corresponds to the 2nd order logic sentence:

$$\forall y.y \rightarrow (\forall x.x \rightarrow y) \rightarrow y \tag{6.5}$$

in which all the variables are bound. This is not an accident. In fact, the representation of any expression that contains only bound variables is invariant with regard to the context in which it is read. Our representation scheme for types provides access to the efficient implementation of several predicated on types:

1. The $typeSubstTop$ operation, obtained directly from [63], substitutes a type for a type variable bound within the abstract type.

2. The ty_eqv function takes two types as arguments and returns true when they are equivalent up to renaming/simplyfying constants. For example (when read with context 6.2.1), both 6.6 and 6.7 evaluate to $true$.

$$ty\_eqv\ [TyList]\ [\Pi X.X \to (String \to X \to X) \to X] \tag{6.6}$$

$$ty\_eqv\ [\Pi Y.Y \to (String \to Y \to Y) \to Y]\ [\Pi X.X \to (String \to X \to X) \to X] \tag{6.7}$$

3. The $is\_constructor\_for$ function (pseudo-code: Program 6.2.1) indicates if objects of a given type can be used to construct objects of another type B using only functional application. For example, both 6.8 and 6.9 below evaluate to true while 6.10 evaluates to false.

$$is\_constructor\_for\ [Int]\ [Boolean \to Int \to Int] \tag{6.8}$$

$$is\_constructor\_for\ [Boolean]\ [(\Pi X.(Boolean \to X) \to X \to X) \to Boolean] \tag{6.9}$$

$$is\_constructor\_for\ [Int]\ [Int \to Int \to Boolean] \tag{6.10}$$

Program 6.2.1 The $is\_constructor\_for$ function

```
is_constructor_for constructed_ty function_ty =
    match function_ty with
     | _ when ty_eqv constructed_ty function_type --> true
     | TyArr _  tyRight --> is_constructor_for constructed_ty tyRight
     |_ --> false
```

4. The is_data_type function (pseudo-code: Program 6.2.3) indicates if a type is a data type as described in section 5.4.2.

5. The $type\_path$ function, (pseudo-code: Program 6.2.2) indicates if a type A is such that an object of type A can be used as an application step to produce an object of type C. For example, both 6.11 and 6.12 below evaluate to true while 6.13 evaluates to false.

$$type\_path\ [Boolean \to Int \to Int]\ [Int] \tag{6.11}$$

$$type\_path\ [\Pi X.(Boolean \to X) \to (X \to X) \to X]\ [Boolean] \tag{6.12}$$

$$type\_path\ [Int \to Int \to Boolean]\ [Int] \tag{6.13}$$

Program 6.2.2 The $type\_path$ function

```
type_path ty_beg ty_end =
   match (is_constructor_for ty_end ty_beg, is_data_type ty_beg) with
     | (true, _)    --> true
     | (_, true)    --> true
     | _            --> false
```

Program 6.2.3 The is_data_type function

```
is_data_type ty =

    let inner_ctx TyVar tyLeft tyRight =
    match (is_constructor_for tyVar tyLeft, tyRight) with
     |false, _ --> false
     |true, _ when ty_eqv tyVar tyRight --> true
     |true, (TyArr (tyLeft, tyRight))--> inner tyVar tyLeft tyRight
     |_->false in

     match ty with
```

Comment 6.2.1 The next subcase deals with the case where the variable is a name for a composed type

```
      |TyVar index when is_name_for_composed_type index -->
        is_data_type (get_composed_type index)
```

Comment 6.2.2 if the type is an abstract type, then it might be the type of a data type

```
    |TyAll (TyArr (tyLeft, tyRight)) -->
       let tyVar = the name of the bound
         type variable within the inner context of ty
       in inner_ctx tyVar tyLeft tyRight
```

Comment 6.2.3 if none of these two cases hold, then the type is not a data type

```
    |_->false
```

6.2.3 Representation of Terms

Terms are represented using one of 9 constructors. The exact definition is:

Definition 6.2.3 ($< Term >$)

```
<Term>  ::= | TmVar <Int>
            | TmAbs <Type><Term>
            | TmApp <Term><Term>
            | TmTAbs <Term>
            | TmTApp <Term><Type>
            | TmString of  string
            | TmFloat <Float>
            | TmInt <Int>
            | TmUnit
```

In which the $TmVar$ constructor points to an entry in the term's associated context. Both the $TmAbs$ and the $TmTAbs$ constructors result in context shifting operations. For example, the term 6.14:

$$\Lambda X.\lambda x^{X}.\lambda y^{X \to X}.y\ x \tag{6.14}$$

is represented as:

```
TmTAbs(
  TmAbs (TyVar 0)
    (TmAbs (TyArr (TyVar 1) (TyVar 1))
      (TmApp (TmVar 0) (TmVar 1))))
```

and is invariant with respect to the context in which it is read. This is not the case for the term 6.15:

$$\lambda x^{TyInt}.plus\ x\ 4 \tag{6.15}$$

which, when read with context 6.2.3 (where $plus$ has index 1) is represented as:

```
(TmAbs TyInt
    (TmApp
        (TmApp (TmVar 2) (TmVar 0))
        (TmInt 4)))
```

because the indexes of the names defined in the context are shifted by the number of binders in an expression, so the term $(TmVar\ 2)$, points to the name $plus$ of context 6.2.3 as it is shifted by 1 binder because of the preceding $TmAbs$ constructor.

6.3 GoalType

context 6.3.1 An example context for an ABGP system that evolves boolean type functions

Type Name	**Type**
$[TyBool]$	$[\Pi X.X \to X \to X]$
Term Name	**Term**
$true$	$\Lambda X.\lambda x^{X}.\lambda y^{Y}.x$
$false$	$\Lambda X.\lambda x^{X}.\lambda y^{Y}.y$

GoalType is the type of the program that the system is set-up to evolve. For example, **GoalType** is:

$$[TyBool \to TyBool \to TyBool]$$

in a system that evolves binary boolean operators (such as xor) within context 6.3.1. A system that evolves programs that take a list of integers (type $[(\Pi X.X \to (Int \to X \to X) \to X)]$) as argument and return its length, will have its **GoalType** set up to be:

$$[(\Pi X.X \to (Int \to X \to X) \to X) \to Int]$$

GoalType is a partial specification of the programs produced by the system. It specifies:

1. The arguments of the programs constructed by the system

2. The type of the object returned by the programs constructed by system

As a rule, in this work, the first argument of **GoalType** will always be a data type. That is, there are types $[A]$ and $[B]$ such that both 6.16 and 6.17 always hold:

$$(ty\_eqv\ \textbf{GoalType}\ [A \rightarrow B]) = true \tag{6.16}$$

$$(is\_data\_type\ A) = true \tag{6.17}$$

6.4 ALLELES AND GENE POOL

An *allele* is a structure that aggregates a normalized System F term with some other information. Alleles are stored in the ecosystem's *gene pool* where they are ordered by the type of the term they enclose. *There is one and only one gene pool for the whole ecosystem*. An allele includes an id field, unique among the other alleles of the same type in the gene pool. An allele also includes a selection probability, stored in its $selectionProb$ field. New alleles are generated by mutating existing alleles, either:

1. as part of a random gene creation cycle, in which an allele is selected for mutation as a *seed* by roulette wheel selection in proportion to its $selectionProb$ value.

2. as a by-product of a crossover operation between two genotypes (see 6.9.2.1)

An allele's $useInEcosystem$ field counts the number of genotypes by which it is carried and its $geneFitness$ field records the average fitness of the genotypes that carry it. Together, these two values are the main factors used to compute $selectionProb$ in the general case. The only exception to this is in the creation of the first generation, as the gene pool is created before the genotypes of the ecosystem. In general, $selectionProb$ relates to the usefulness of the allele as a building block in the ecosystem. The formal definition of an allele is given below:

Definition 6.4.1 ($< Allele >$)

```
<Allele> ::= {
        expression : <Term>;
        id : <Int>;
        selectionProb : <Float>;
        useInEcosystem : <Int>;
        geneFitness : <Float>;
    }
```

6.4.1 Allele Complexity

The random process by which alleles are produced implies that some new alleles may be quite large. There is a parsimony measure which is used to limit the size of an allele. It corresponds to the *complexity* of the term it expresses. This measure is arbitrary, but its purpose is to provide a sense of the size of the term that an allele contains. This allows an ABGP system to reject alleles that it considers too large.

Definition 6.4.2 (Expression Complexity) The complexity of an expression is defined as:

1. Type constants and type variables have complexity 1.
2. Arrow types have the complexity of their left type plus the complexity of their right type, plus 1.
3. The complexity of an abstract type is the complexity of the body of its abstraction plus 1.
4. The complexity of a term constant or variable is 1.
5. The complexity of an application term is the sum of the complexities of its two arguments plus 1.
6. The complexity of an abstract term is the complexity of the body of the abstraction plus 1.
7. The complexity of a term abstracted by type is the complexity of the body of the abstraction plus 1.
8. The complexity of a term/type application (such as $((\Lambda X.\lambda x^X.x)\ [Int])$) is the sum of the complexities of the term on the left side and of the type on the right side.

In the implementation, the complexity of an expression is computed by the function $complexity\_of\_expr$. $maxGeneComp$ is the value that specifies an upper-bound on the size of the alleles that may be included in the gene pool of an ABGP system. It is provided to the system at the time of the system's initialization by the system's designer. When new alleles are created, they may only be included in the gene pool if their size is smaller than $maxGeneComp$. As it turns out, one of the results that was obtained during the course of this research is that this value is crucial to the performance of an ABGP run (see 7.6.1).

6.4.2 Gene Pool

As mentioned previously, alleles are stored in a *gene pool*, which is a wrapper for a hash table in which the keys are types and the values are lists of alleles that contain expressions of the same type as the key index that indexes the lists. Only one copy of an allele may exist in the gene pool at any one time. The definition of a gene pool is below:

Definition 6.4.3 (Gene Pool)

```
<GenePool> := (<Type>, <Allele> list) Hashtbl
```

6.4.3 The Gene Pool's Kernel

Given a context C and a **GoalType**, the kernel of the gene pool is constructed thus:

1. The typed free variables and the names for System F terms in C are entered in the gene pool as new alleles, with their types as corresponding genes.

2. A series of constructors for the type **GoalType** are produced and entered in the gene pool. The constructors are produced using the $make\_constructors$ function. The constructors are automatically deduced from **GoalType** using the $make\_constructors$ function. They are included in the original gene pool as functions guaranteed to result in the production of an object of type **GoalType**. The inclusion of such functions facilitates the discovery process of new species.

We do not include the pseudo-code for the $make\_constructors$ function as it is a rather complex function that uses context shifting and translation operations on types which are of little interest in the present discourse (the actual OCaml implementation code is given in appendix B). However, as it is a central function of the system, we will describe what it does. The function takes a type $[C]$ that has the form $[A \rightarrow B]$ in which $[A]$ is a data type (in the required format) and outputs a series of functions that may be used to construct objects of type $[C]$. The functions are created using a fresh bound variable x of type $[A]$ and applying $(x\ [B])$ to successive new bound variables to yield an object of type $[B]$. This object is then used as the inner body of a series of functional abstractions built using the possible permutations of the variables created during the process. For example, when given the following type as input:

$$[TyBool \rightarrow Int]$$

the function outputs the following constructors:

$$\lambda x^{Int}.\lambda y^{Int}.\lambda a^{TyBool}.a\ [Int]\ x\ y$$
$$\lambda y^{Int}.\lambda x^{Int}.\lambda a^{TyBool}.a\ [Int]\ x\ y$$

Output 6.4.1 and 6.4.2 provide two more examples of the function's behavior, obtained using the context 6.2.1:

6.4.4 Gene Diversity

Using the OCaml built-in $hash$ function in conjunction with the nameless representation style, it is relatively easy to compare terms by structure. What we compare in this case are alleles, not genes. We only need to compare alleles if they have the same type (alleles of the same gene). The function $hash$ takes a parameter $depth$ up to which it compares structures. If two structures are similar (up to $depth$), the function returns the same value for both. We applied this to expressions as a method to differentiate alleles from each other. Each allele's expression-dependent hash value is computed, and alleles that are too similar to existing alleles are refused entry into the gene pool. We decided to use this feature as we realized that the diversity of the genes contained in the gene pool had an impact on the performance of the system. As a result, we incorporated a global user input parameter $geneDifference$ into the design of the system. The value $geneDifference$ is used as a parameter to the OCaml $hash$ function which is then used to compare new genes to existing genes in the gene pool. When this value is higher, a gene needs to be "more different" than the other genes in the gene pool in order to be included in the gene pool. The impact of this parameter is discussed in section 7.6.2. The down-side of using this method to encourage genetic diversity is that the cost of the computation of the hash value for each allele is high and that the higher the precision to which it is set, the more new genes need to be produced to fill up the gene pool as more genes are refused entry.

output 6.4.1

```
The type: [(TT T. T-> (TyBool-> T)-> (Int-> T)-> T)-> Int] is
constructible by the following 6 terms:

    1. lam m:TyBool-> Int.lam q:Int.lam r:Int-> Int.
    lam q':TT G. G-> (TyBool-> G)-> (Int-> G)-> G.q' [Int] q m r

    2. lam d:TyBool-> Int.lam i:Int-> Int.lam b:Int.
    lam b':TT S. S-> (TyBool-> S)-> (Int-> S)-> S.b' [Int] b d i

    3. lam m:Int-> Int.lam d:TyBool-> Int.lam i:Int.
    lam k:TT N. N-> (TyBool-> N)-> (Int-> N)-> N.k [Int] i d m

    4. lam j:Int-> Int.lam u:Int.
    lam y:TyBool-> Int.
    lam o:TT S. S-> (TyBool-> S)-> (Int-> S)-> S.o [Int] u y j

    5. lam c:Int.lam g:Int-> Int.lam r:TyBool-> Int.
    lam y:TT G. G-> (TyBool-> G)-> (Int-> G)-> G.y [Int] c r g

    6. lam r:Int.lam b:TyBool-> Int.lam n:Int-> Int.
    lam l:TT R. R-> (TyBool-> R)-> (Int-> R)-> R.l [Int] r b n
```

output 6.4.2

```
The type: [TyList-> Int-> String] is constructible by the following
2 terms:

    1.lam c:String-> (Int-> String)-> Int-> String.
    lam i:Int-> String.lam a:TyList.a [Int-> String] i c

    2. lam m:Int-> String.
    lam t:String-> (Int-> String)-> Int-> String.
    lam x:TyList.x [Int-> String] m t
```

6.4.5 Generating New Gene Pools

Each new generation's gene pool, including the first generation, is constructed by mutating alleles that are already in the gene pool to produce new alleles. The system-wide variable $maxComp$ defines a ceiling on the sum of the complexity of the alleles that are in the gene pool at any one time. The terms defined in the system's terminal set (as defined by the GP's context) and the constructors derived from **GoalType** are used to build the kernel of the first gene pool which is then constructed to $maxComp$ as any other gene pool would be. Each allele is assigned a probability of being selected as a seed. The probability of being selected as a

seed is set for each element of the gene pool's kernel as:

$$\frac{1}{n} \tag{6.18}$$

where n is the initial number of allele obtained by the direct translation of the expressions in the GP's context and of the **GoalType** constructors into alleles. For the next generations, the probabilities are calculated as functions of the fitness of the genotypes that carry the alleles. The probability is stored in the $selectionProb$ field of each allele. Each allele is chosen as a seed with probability $selectionProb$ and a set of descendent alleles is thus produced. Before insertion, the probability p of the seed allele is divided by $d+1$, were d is the cardinality of the set of its descendent alleles, to yield its new selection probability, as well as the selection probability of each of its descendants. This scheme is designed to encourage diversity optimization. For example, given the context of table 6.3.1, the initial set of alleles is created as:

```
type: TyBool
 id:1 - exp:true - selectionProb:0.5
 id:0 - exp:false - selectionProb:0.5
```

Giving each of the two original alleles of the $[TyBool]$ gene an equal chance of being chosen as seeds. Following the initial construction of the gene pool, if 4 new alleles are constructed by mutating the allele that expresses $true$, the new selection probabilities will be set as:

```
type: TT D. (D-> D-> D)-> D-> D-> D
 id:0 - exp:Lam V.lam k:V-> V-> V.lam a:V.k a - selectionProb:0.1
type: TT Q. (Q-> Q)-> (TyBool)-> Q-> Q
 id:0 - exp:Lam M.lam d:M-> M.lam e:TyBool.lam u:M.e [M] (d u) u - selectionProb:0.1
type: TT M. (TT H. H-> (M-> H-> H)-> H)-> M-> (M-> M-> M)-> M
 id:0 - expression:Lam T.lam w:TT W. W-> (T-> W-> W)-> W.w [T] - selectionProb:0.1
type: TT L. L-> L
 id:0 - exp:Lam T.lam a:T.a - selectionProb:0.1
type: TyBool
 id:1 - exp:true - selectionProb:0.1
 id:0 - exp:false - selectionProb:0.5
```

As there are now 5 alleles sharing the initial selection probability 0.5 of the allele that expresses $true$. In this example, when the next round of gene creation occurs, the allele expressing $false$ is 5 times more likely than any other allele in the gene pool to be chosen as the next seed. Appendix D.2 lists the elements of a gene pool randomly generated from the primitives defined in table 6.2.3.

6.4.6 Generating New Alleles

When a new set of alleles is created by seed mutation, the alleles from the descendant set that are not already in the gene pool and that do not violate size constraints (see section 6.4.1) are added to the gene pool. There are many possible mutation operations that may be applied to a seed to generate a set of descendant genes. For example:

1. If the seed is an allele whose $expression$ field contains the term t, of type $[\Pi X.V]$, then one can generate the descendant set of all alleles with expression $(t\ A)$, where A is a type that exists in the gene

pool. The $geneId$ field for each of these new alleles is assigned by the system, and we describe the process by which a value is assigned to the $selectionProb$ field in 6.4.5.

2. If the seed has its $expression$ field t, of type $[A \rightarrow B]$, then one can generate the descendant set of alleles with expression $(t\ a)$, where a is a gene expression of type A that exists in the gene pool.

3. If the seed has its $expression$ field t, of type $[T]$, that contains a constant a^A, then one can generate the descendant set (of one element) of alleles with expression $(\lambda x^A.t\{a := x\})$ of type $[A \rightarrow T]$

4. If the seed has its $expression$ field t, of type $[T]$ that contains a type constant $[A]$, but does not contain any term constant of type $[A]$ then one can generate the descendant set (of one element) of alleles with expression $(\Lambda X.t\{A := X\})$ of type $[\Pi X.T\{A := X\}]$

There are many other possibilities. The set of rules that we use in this research was obtained by trial and error. We list these mutation rules in appendix C, but they include the four listed above.

6.5 SPECIES

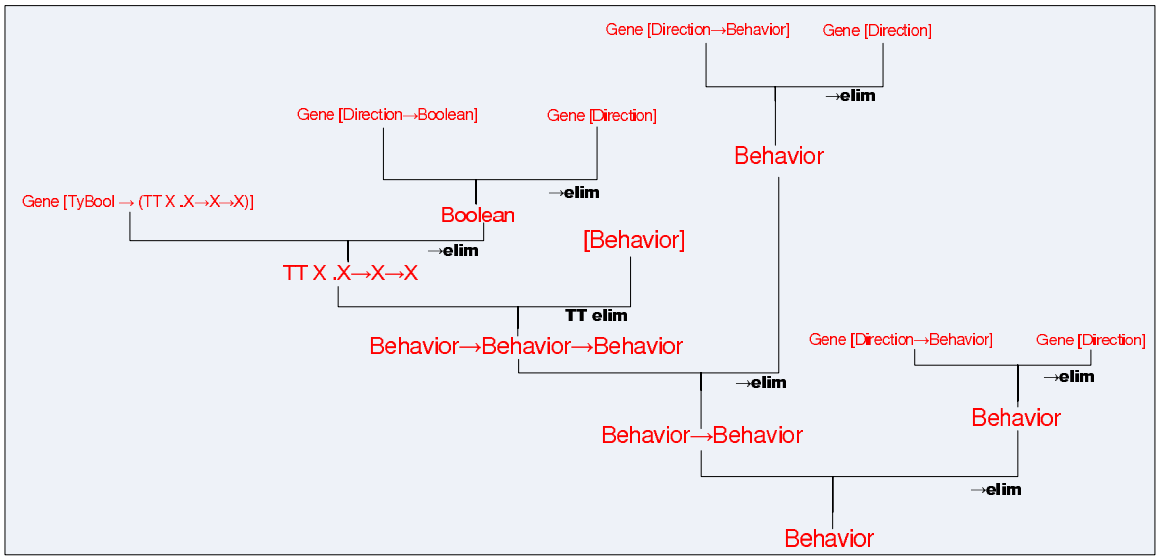

Figure 6.2: A species constructed on a gene pool generated from the primitives defined in table 6.2.2 with **GoalType** $[Behavior]$

Each genotype in the population corresponds to a program of type **GoalType**. **GoalType** guides the construction of species. Species are 2^{nd}-order logic constructive proofs built using the previously defined rules of $\rightarrow$-elim and Π-elim. Genes in the gene pool (the keys of the lists of alleles) are used as propositions in the proofs. The result is a specific arrangement of alleles. When all the gene locations of a species are occupied by the appropriate alleles, a *genotype* is obtained. A genotype of a species is the result of assembling specific alleles of a gene as specified by the genotype's species. Figure 6.2 is the tree representation of a species for a GP system that evolves a function of type $[Behavior]$ using the set of primitives defined in table 6.2.2. Figure 6.3 shows the pattern specified by a species for a program of type $[Int \rightarrow Int]$. A genotype of

this species is obtained by filling in specific alleles for each genes for genes 1 through Gene 6. The formal definitions of the operators used to construct species is below.

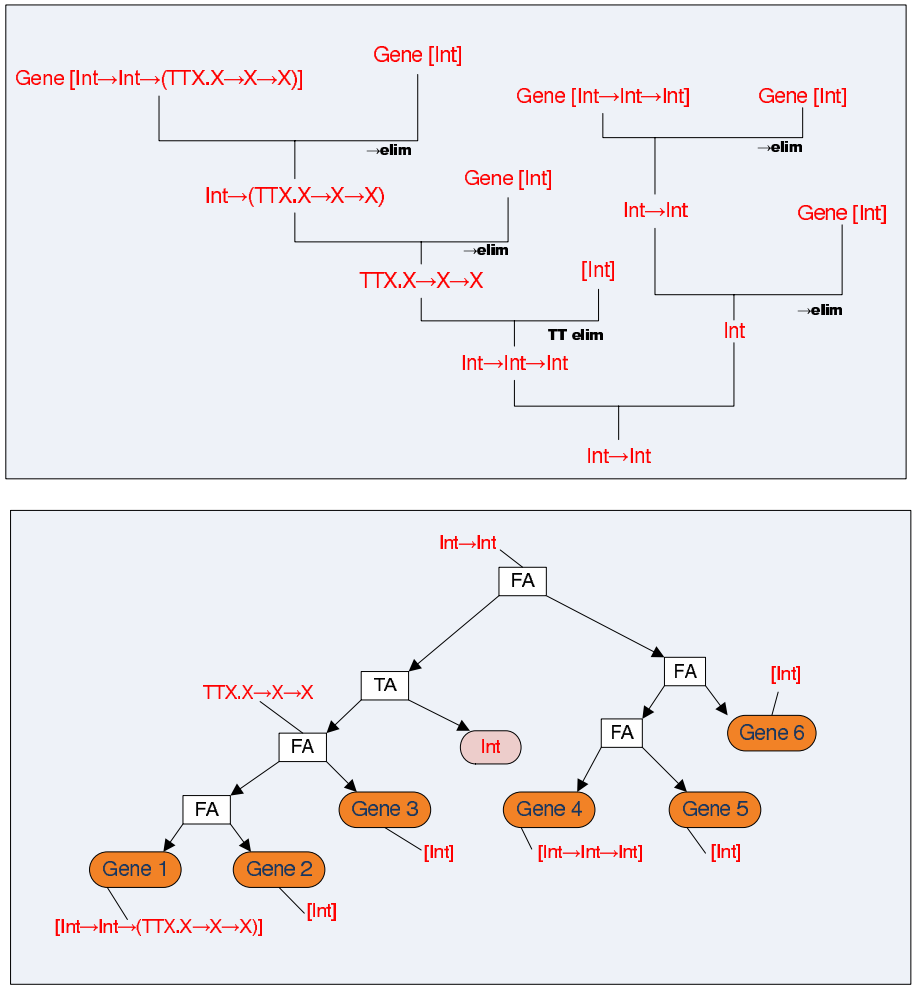

Figure 6.3: A species (above) and the genotype pattern that it produces (below)

Definition 6.5.1 (Species)

```
<Species> ::=    | "Gene" <Ty>
                 | "TTelim" <Species> <Ty>
                 | "->elim"  <Species> <Species>
                 | "Partial" <Ty> <Species>
```

Partial proofs are used during the process of generating new proofs. Below is an example of a species coding behavioral pattern for an ant system:

```
<->elim
    <->elim
        <TTelim <Gene[TT X.X->(Object->X)->X]><[Behavior]>>
        <Gene[Behavior]>>
    <Gene [Object->Behavior]>>
```

The habitual tree-like version of this species is represented in the upper left portion of figure 6.6. The next section describes the generation process for new proofs.

6.5.1 Generating Species

The random generation of new species was one of the major problems that had to be resolved for the completion of this work. This problem essentially reduces to randomly producing a set of unique proofs of a given proposition. Species are either grown using a top down approach, or created from the mutation of an existing species. A maximal depth value on proofs $maxProofDepth$ is defined.

The first step to the random generation of a set of new species is the identification of genes that may be used as propositions by the species. This is done using the previously defined (6.2.2) $type\_path$ function.

6.5.1.1 Generating an Initial Set of Potential Proofs

The function $initialProofSet$ uses the $type\_path$ function to create a list of proofs of a given type. Each proof generated by $initialProofSet$ is either a proof in the form ($Gene$ $[ty]$) or a partial proof. The pseudo-code for

Program 6.5.1 The $initialProofSet$ function

```
initialProofSet [goal_type] genes =
```

Comment 6.5.1 $candidateTypes$ is extracted as a list of types $[ty]$ in genes such that ($type\_path$ $[ty]$ $[goal\_type]$) holds;

```
let candidateTypes  =
    get_alleles genes (fun ty -> type_path ty goal_type)

in map candidateTypes
(fun ty ->
    match ty with
    | _ when ty_eqv ty goal_type --> Gene ty
    | TyAll sub_ty --> Partial (goal_type (TTelim (Gene ty) goal_type))
    | _ --> Partial (goal_type (Gene ty)))
```

the function is given in 6.5.1. In it, we note the use of the following two functions (which will be seen again in other code listings):

1. the $get\_alleles$ function takes a gene pool and a predicate on genes as arguments and returns a list of all the alleles in the gene pool that are alleles of a gene that satisfies the argument predicate.
2. the map function takes a list of objects of type A and a function f of type $A \rightarrow B$ as arguments and returns the list resulting from the application of f to each element of the argument list.

Figure 6.4 illustrates a call to the $initialProofSet$ function with the goal type argument $[Int]$. In the case illustrated by the figure, the call returns four proofs, three of which are partial and one is complete. Given

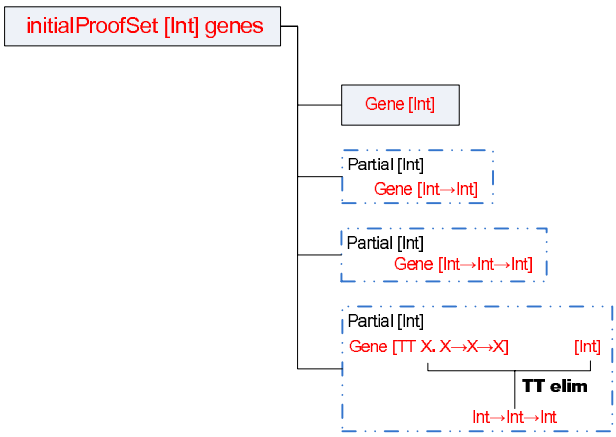

Figure 6.4: A call to the $initialProofSet$ function with goal type $[Int]$

GoalType, the call:

$$initialProofSet \textbf{ GoalType } genes$$

in which $genes$ is the name for the gene pool associated with the GP, provides a starting point to the random generation of our initial set of proofs. Following this starting point, each proof in the list is separately passed as argument to the $modifyProof$ function which takes a (possibly partial) proof as argument and returns a list of (possibly partial) proofs generated from its modification. $modifyProof$ uses the get_type function which computes the proposition (or type) that a proof proves. The pseudo-code for the get_type function is provided in program 6.5.2.

Given a partial proof in the form $(Partial\ [Ty]\ proof)$, we may suppose that the following statements always hold true:

Program 6.5.2 The get_type function

```
get_type proof =
   match proof with
   | Gene ty --> ty
   | TTelim sub_proof ty --> typeSubstTop (get_type sub_proof) ty
   | ->elim proof1 proof2 -->
        match ((get_type proof1),(get_type proof2)) with
        |((TyArr tyLeft tyRight) , tyArg)
            when (ty_eqv tyLeft tyArg) --> tyRight
        | _--> fail (proof is mal formed)
   | Partial ty _ --> ty
```

1. $(type\_path\ (get\_type\ proof)\ [Ty])$ is always true
2. Given a proof $p = (Partial\ [Ty]\ proof)$, where $(ty\_eqv\ [Ty]\ (get\_type\ proof))$ is true, then p may be replaced by simply be replaced by the proof $(proof)$ of type $[Ty]$ in any context.

We are now almost ready to provide the pseudo-code for $modifyProof$. The remaining discussion point pertains to the stopping condition of the process. We may be able to generate hundreds of proofs if we let the function go on for long enough, but we only need to generate enough species to produce enough genotypes for our first generation. As we haven't introduced a formal definition for genotypes yet (we do this in the next section), we content ourselves to using a call to the function $enough\_genotypes$ which returns true when there are enough proofs to construct the required number of genotypes. $enough\_genotypes$ lets the $modifyProof$ function know when it has produced enough proofs to generate the first generation of the ecosystem eco. The $modifyProof$ function also uses a $depth$ numeric argument, which is the maximum depth in the proof tree at which $modifyProof$ may still attempt to modify existing proofs. We should note that $modifyProof$ might generate proofs that already exist, in which case, they are excluded from the resulting set of species. The commented pseudo-code is provided in listing 6.5.4. $modifyProof$ uses the $cartesianProduct$ function defined in 6.5.3 with signature:

$$cartesianProduct : (A \rightarrow B \rightarrow C) \rightarrow A\ list \rightarrow B\ list \rightarrow C\ list$$

The $cartesianProduct$ function produces a list of all the combination possibilities of two lists of objects using a combination function. Figure 6.5 illustrates the path that the algorithm takes when it is given $[Int]$ as

Program 6.5.3 The $cartesianProduct$ function

```
cartesianProduct combineFunc alist blist =
   let comb a = map (fun b -> combineFunc a b) blist in
   List.flatten (List.map (fun a -> comb a) alist)
```

its goal type argument, with $Gene[Int]$ as its starting proof.

6.6 GENOTYPES

Genotypes are arrangements of alleles. The arrangement is specified by the species to which a genotype belongs. A genotype in ABGP is basically a tree in which the leaves may be either the id of alleles that exist in the gene pool or types.

Definition 6.6.1 (Genotype Definition (ABGPS))

```
<genotype> ::=  | "AlleleId" <int>
                | "TA" <genotype> <ty>
                | "FA" <genotype> <genotype>
```

It is clear from the definition that a genotype on its own doesn't do much. A genotype can only be interpreted as a computer program in the wider context of the gene pool with which it is associated and of the species to which it belongs. Knowing which species a genotype belongs to and having the gene pool to which it relates is the only way to know which alleles are referred to by the "AlleleId" part of the genotype. As a consequence, whenever a function is called that uses a genotype as its argument, that function will usually also have a gene pool and a species as complementary arguments. Figure 6.6 represents four genotypes that belong to the species defined in section 6.5. Given a species $spec$ and a gene pool $genes$, the function

Program 6.5.4 The $modifyProof$ function

```
modifyProof goal_type genes proof depth  =
match depth, proof with
```

Comment 6.5.2 The following two sub-cases handle simplification of partial proofs

```
    |(_, (Partial ty subproof)) when  ty_eqv (get_type subproof) ty -->
        modifyProof goal_type genes subproof depth
    | (_, (Partial _ partial_proof)) -->
        match get_type partial_proof with
                |TyArr ty1 ty2 ->
                    map (initialProofSet ty1 genes)
                    (fun pr -> modifyProof goal_type genes
                    (Partial (goal_type, ImplyElim (partial_proof, pr))) (depth -1))
                |TyAll ty -->
                    modifyProof goal_type (Partial (goal_type,
                    (TTelim (partial_proof, pr)))) (depth -1)
```

Comment 6.5.3 The next sub-case's position in the list of clauses insures that it will not be a partial proof. This case detects the argument as a complete proof, wraps it in a list and returnes it as result

```
    | _ when ty_eqv (get_type proof) goal_type --> (proof::Nil)
```

Comment 6.5.4 Next sub-case handles maximum depth reached. Can't go deeper

```
    | 1, _ ->  initialProofSet goal_type genes
```

Comment 6.5.5 Next sub-case: argument proof is a gene, but maximum depth is not reached, so the node is branched out again.

```
    |_ , (Gene ty) --> initialProofSet ty genes
```

Comment 6.5.6 sub-case proof is $\rightarrow$-elim

```
    |_, (->elim proof1 proof2)-->
            concatenate (initialProofSet goal_type)
                (cartesianProduct
                    (fun pr1 pr2 -> ImplyElim(pr1,pr2))
                    (modifyProof (get_type proof1) genes proof1 (depth - 1) )
                    (modifyProof (get_type proof2) genes proof2 (depth - 1) ))
```

Comment 6.5.7 sub-case proof is Π-elim

```
    | _, (TTelim subproof ty)-->
                concatenate (newProofs goal_type)
                (map (modifyProof (get_type subproof) genes subproof (depth - 1))
                (fun pr -> ForAllElim(pr,ty)))
```

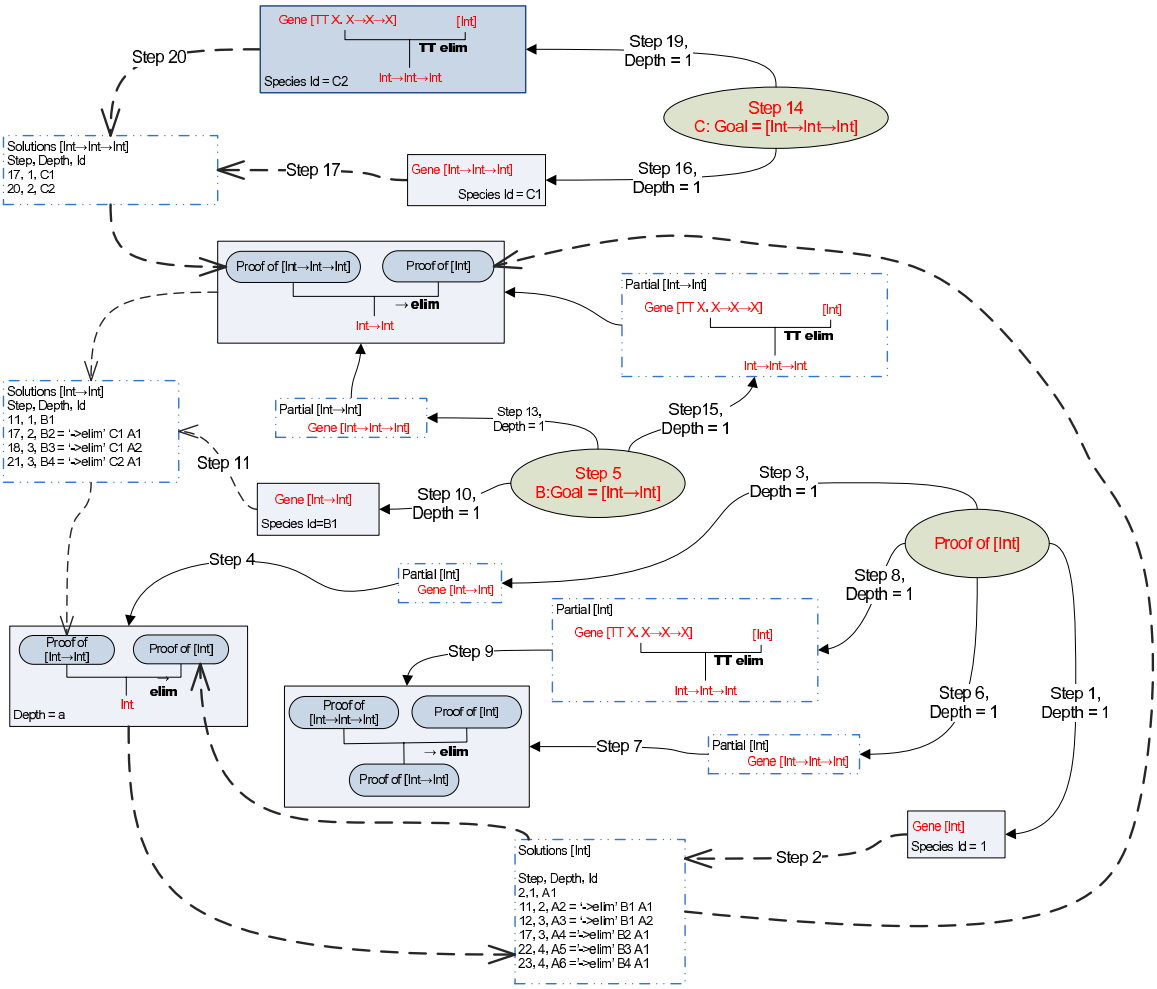

Figure 6.5: A trace of the $modifyProof$ (code 6.5.4) function when initial argument is $[Int]$

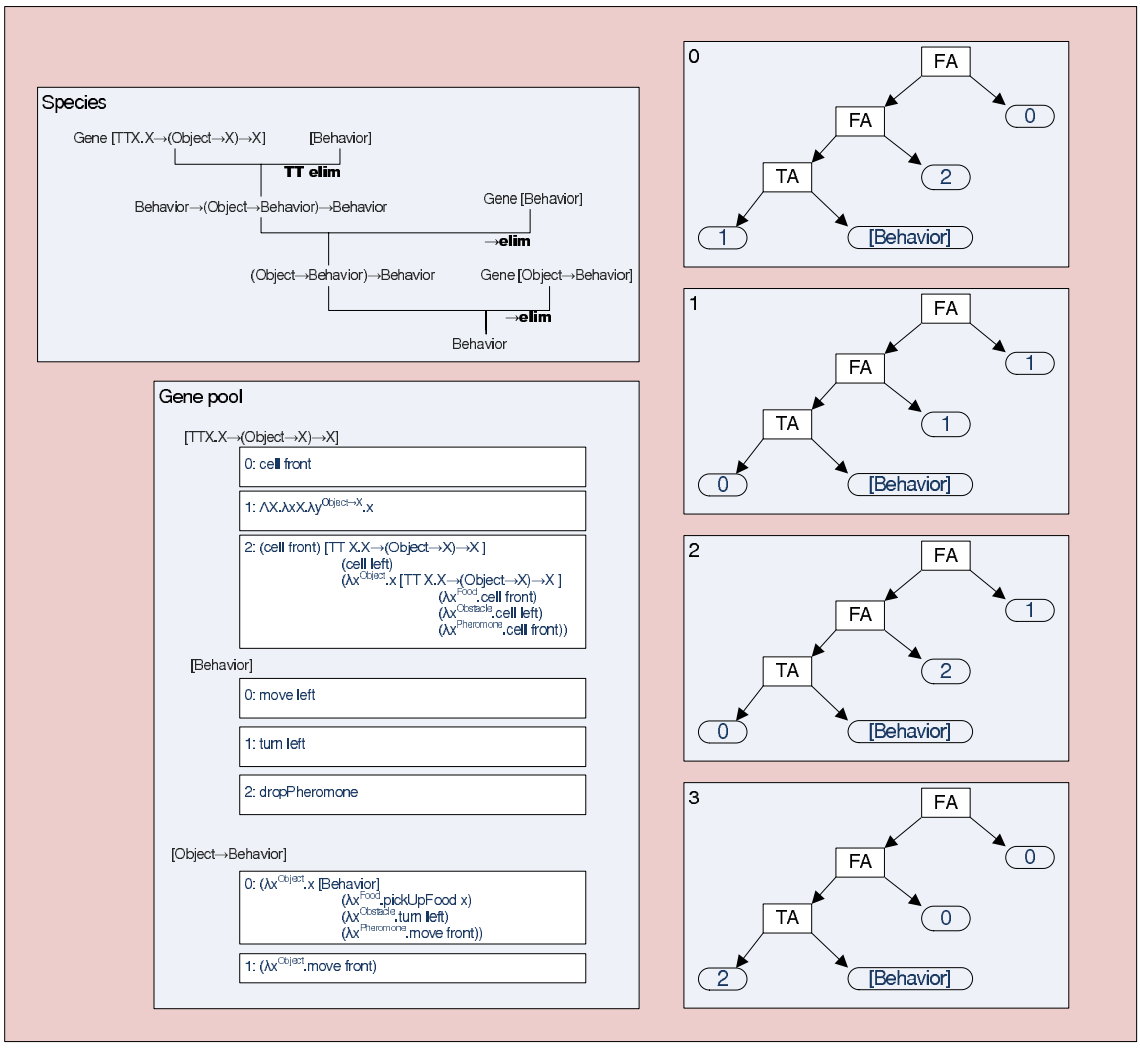

Figure 6.6: Four genotypes that belong to the same species and the subset of the gene pool that contains the alleles that compose them

$genotypeNum$ (listing 6.6.1) returns the number of different possible genotypes that may be constructed. Within $genotypeNum$, the functions $tyMem$ and $tyFind$ on $genes$ are used. The $tyMem$ function takes a type and a gene pool as arguments and answers true if the type is a gene in the gene pool, while $tyFind$ takes a gene (which is a type) and a gene pool as argument and returns the list of alleles associated with the gene. Moreover, given a species $spec$ and a gene pool $genes$, a simple modification to the $genotypeNum$ function yields the $all\_genotypes$ function which creates all the possible genotypes that belong to $spec$. As genotypes are often created in such groups, they are differentiated in status within the system by being associated with a $genoStatus$ object. Objects of type $genoStatus$ are defined as:

```
<genoStatus> ::=  | "Dead" <float>
                  | "Alive" <float>
                  | "Unborn"
```

Program 6.6.1 The $genotypeNum$ function

```
genotypeNum genes spec =
```

Comment 6.6.1 The $tnum$ sub-routine, below, returns the number of alleles of a gene $[Ty]$

```
let tnum [Ty] =
    match tyMem genes [Ty] with
    |true --> List.length (tyFind genes [Ty])
    |false --> 0 in

match spec with
| Partial _ --> 0
| Gene [Ty] --> tnum [Ty]
| TTelim sub_spec _ --> genotypeNum genes sub_spec
| ->Elim sub_spec1 sub_spec2 -->
    (genotypeNum sub_spec1) * (genotypeNum sub_spec2)
```

This corresponds to the three possible state in which a genotype may exist:

- An unborn genotype is a genotype that hasn't been evaluated yet.
- A genotype is marked as 'Dead' to indicate that it has been evaluated, but has no more role to play in the production of new generations and will never again be selected for crossover.
- A genotype marked as 'Alive' may be used as a selection tool during the construction of the next generation. At the beginning of each generation (except the initial one), there are $ecosystemDefaultSize$ genotypes marked as 'Alive' in the system. $ecosystemDefaultSize$ is the input parameter that defines the size of the ecosystem in terms of programs to evaluate during the current generation.

6.6.1 Populations

The associations between genotypes and their status is recorded within a $Population$ object. Each species (proof) is associated with a unique $Population$ structure and an ecosystem encasulates several $Population$ structures. The total number of individuals marked as 'Alive' in the whole ecosystem is defined by the parameter $ecosystemDefaultSize$. An object of type $Population$ is defined as:

```
<Population> ::= {
     pattern : <Species>;
     fitness : <float>;
     genotypes : (<genotype>, <genoStatus>) Hashtbl;
     }
```

We also make use of the function: $alive\_population : Population \rightarrow \mathbb{N}$ to obtain the number of genotypes marked as 'Alive' in a population.

6.6.2 Generating New Genotypes

Given a species $spec$ and a gene pool $genes$, we may create n genotypes, provided that:

$$(genotypeNum\ genes\ spec) - (alive\_population\ pop) \geq n$$

holds, where pop is the population associated with $spec$. The function:

$$birth\_genotypes\_of\_species\ genes\ spec\ n$$

results in n new genotypes of the species $spec$ being marked as 'Alive'. These genotypes may be found in the set of previously created genotypes marked as 'Unborn' or may be created outright. The $birth\_genotypes\_of\_species$ function implements an effort to maximize the genetic diversity of the population associated with $spec$. This is done by assigning a temporary counter to each allele in $genes$ that are or may be used to construct organisms of the species $spec$. The counters are incremented as new live genotypes are added to the population. The idea is to make each allele's probability of being selected to be part of a new live genotype inversely proportional to the number of times it has already been chosen for that purpose. To do this, the system selects the alleles with their counter set at the lowest possible as genes for each new genotypes marked as 'Alive'. In the example below, initial counters are presented for each allele in the gene pool of figure 6.6. For example, at the very beginning of the spawning process, when no genotype is alive, the counters for each allele are set to 0:

```
|Gene Pool Dump 1           |
[TT X.X->(Object->X)->X]
    id:0 : 0
    id:1 : 0
    id:2 : 0
[Behavior]
    id:0 : 0
    id:1 : 0
    id:2 : 0
[Object->Behavior]
    id:0 : 0
    id:1 : 0
```

after the following genotype is switched from 'Unborn' to 'Alive' and is inserted in the population:

```
<FA
    <FA
        <TA <AlleleId 2><[Behavior]>>
        <AlleleId 1>>
    <AlleleId 1>>
```

the counters are set at:

```
|Gene Pool Dump 2           |
[TT X.X->(Object->X)->X]
    id:0 : 0
    id:1 : 0
```

```
    id:2 : 1
[Behavior]
    id:0 : 0
    id:1 : 1
    id:2 : 0
[Object->Behavior]
    id:0 : 0
    id:1 : 1
```

There are now several possibilities for the next new genotype. It might, for example, be:

```
<FA
    <FA
        <TA <AlleleId 1><[Behavior]>>
        <AlleleId 0>>
    <AlleleId 0>>
```

which would set the counters at:

```
|Gene Pool Dump 3           |
[TT X.X->(Object->X)->X]
    id:0 : 0
    id:1 : 1
    id:2 : 1
[Behavior]
    id:0 : 1
    id:1 : 1
    id:2 : 0
[Object->Behavior]
    id:0 : 1
    id:1 : 1
```

As a result of this sequence of operations, the next genotype would have to carry allele 0 from the gene $[\Pi X.X \to (Object \to X) \to X]$ and allele 2 from the gene $[Behavior]$.

6.7 PHENOTYPES

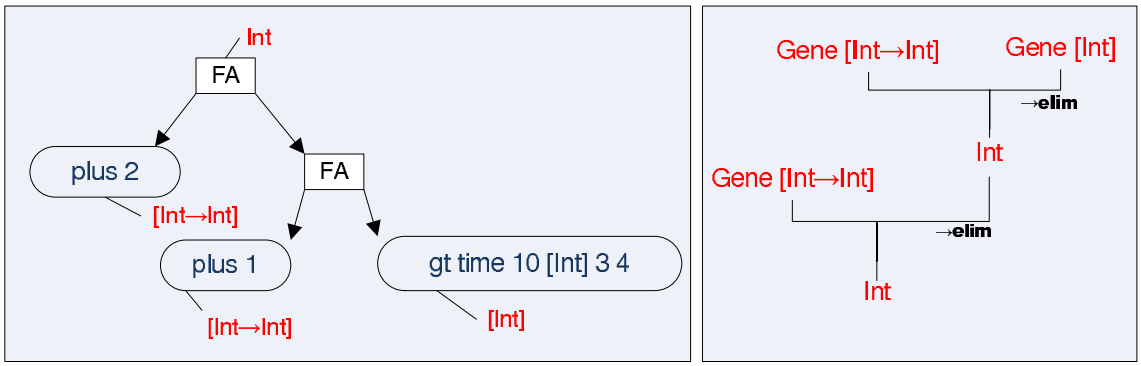

Figure 6.7: A non-evaluated phenotype assembled from alleles contained in a gene pool built using the context defined in table 6.2.3 and the species it belongs to (right)

As per the definitions provided in 2.1, the system's interpreter derives a phenotype from a genotype. The $interpret$ function matches each of the leaf building block of a genotype with the expression of the allele to which it is a pointer and normalizes the resulting expression. Figure 6.7 illustrates the STN tree form of the phenotype:

$$(plus\ 2\ (plus\ 1\ (gt\ time\ 10\ [Int]\ 3\ 4))) \tag{6.19}$$

The function $interpret$ (program 6.7.1) takes a gene pool, a species and a genotype as arguments and returns the phenotype associated with the genotype. The function get_term is used to extract the term expressed by an allele using the allele's id and its type, while $TmTApp$ is used to construct a type application from a term and a type and $TmApp$ is used to construct a term application from two terms. The $normalize$ function does what its name indicates. It is possible to obtain the same phenotype from the conversion of different genotypes, even when they belong to different species. Figure 6.8 illustrates this.

Program 6.7.1 The $interpret$ function

```
interpret genes species genotype =

    let non_normalized_term =
        match (species , genotype) with
        | (Gene [ty], AlleleId id) -->  get_term genes ty id

        | (TTelim species [ty1] , TA genotype [ty2]) when ty_eqv [ty1] [ty2] -->
            TmTApp (interpret genes species genotype) [ty1]

        | ((->elim spec1 spec2) , (FA gtype1 gtype2)) -->
            TmApp (interpret genes spec1 gtype1) (interpret genes spec2 gtype2)

        | _->failwith "Invalid pair proof/program" in

    normalize non_normalized_term
```

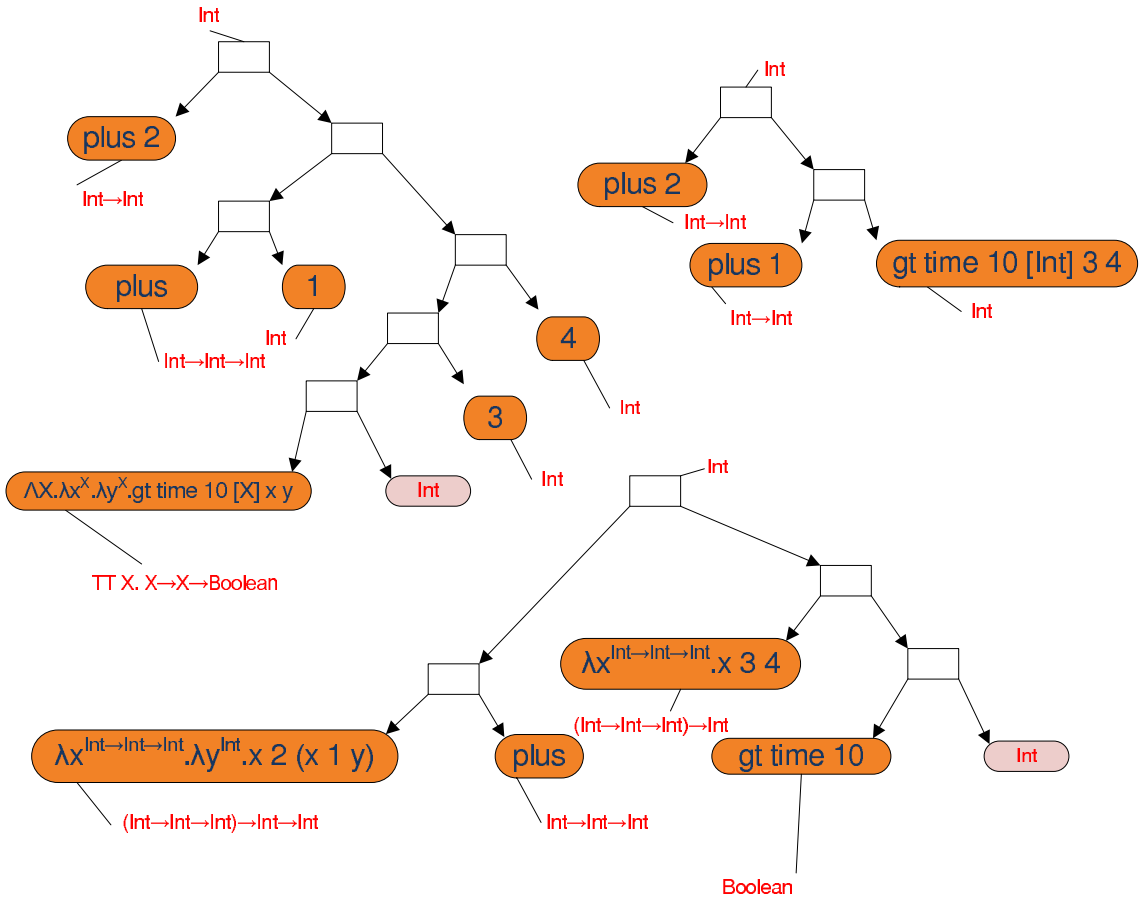

Figure 6.8: The same phenotype results from these three different genotypes as they normalize to the same term

6.8 THE ABGP ECOSYSTEM

In many ways, the ecosystem *is* the ABGP system. It brings all the elements presented so far into one data structure. An object of type *Ecosystem* is a tuple defined as:

Definition 6.8.1 (Ecosystem)

```
Ecosystem  ::= {
      generation_count : <int>;
      context : Binding * string list
      goal_type : <ty>;
      genes : <GenePool> ;
      kingdom : (<Species>,  <Population>) Hashtbl
}
```

The attribute $kingdom$ is a hash table. The keys are objects of type $Species$ and the values are objects of type $Population$. We make use of the the implemented function $alive\_eco : Ecosystem \rightarrow \mathbb{N}$ to obtain the

number of genotypes marked as 'Alive' in an ecosystem, and of $unborn\_eco : Ecosystem \rightarrow \mathbb{N}$ to obtain the number of genotypes marked as 'Unborn' in an ecosystem.

6.8.1 Generating the Initial Ecosystem

1. Init 1. A context is provided to eco, it forms the basis for the gene pool $genes$.

2. The system deduces **GoalType** from provided test cases list or it is explicitly provided by the system's user.

2. GrowPool The gene pool is grown randomly to size $maxComp$ by randomly mutating the genes that are already in the gene pool. No new gene may be larger than $maxGeneComp$ and larger genes are discarded. Moreover, each new allele has to be distinct from the alleles currently in the gene pool by the value $geneDifference$.

3. IsReady? The number of possible genotypes in eco is computed based on the species in the system and the type and number of genes in the gene pool using the $genotypeNum$ function. If it is possible to produce $ecosystemDefaultSize$ genotypes, go to $BuildGeneration$, otherwise go to step $AddSpecies$

4. AddSpecies Produce one new species, either by modifying species already in the ecosystem or by brute search for new proofs of **GoalType** with limited depth, using the genes in the gene pool as propositions. Delete all the alleles in the gene pool that can not be used in any of the species currently in the ecosystem. Go to step $GrowPool$.

5. BuildGeneration The total potential number of genotypes n is computed (using the $genotypeNum$ function). Each species $spec$ is allocated m new live genotypes where:

$$m = \frac{(genotypeNum\ genes\ spec) * ecosystemDefaultSize}{n}$$

The equation above simply computes the number of genotypes to spawn for each species as a proportion of the number of genotypes that can be spawned per species (in relation to the ecosytem's gene pool) in relation to the total number of genotypes that must be spawned in order to obtain $ecosystemDefaultSize$ genotypes in the whole ecosystem.

6.9 From Generation to Generation

Below is an overview of the ABGP algorithm:

1. The user provides the input arguments, the context and the test cases.
2. The system deduces **GoalType** from the test cases and builds the gene pool's kernel.
3. The gene pool is grown randomly to size $maxComp$ by randomly mutating the alleles that are already in it. No new allele may be larger than $maxGeneComp$.
4. The number of possible genotypes is calculated based on the species in the system and the type and number of genes in the gene pool. If it is possible to produce $ecosystemDefaultSize$ genotypes, go to step 6, otherwise go to step 5

5. Produce new species either by modifying species already in the ecosystem or by brute search for new proofs of **GoalType** with limited depth, using the types of the genes in the gene pool as propositions. Delete all the alleles in the gene pool that can not be used in any of the species in the ecosystem. Go to step 3.

6. $ecosystemDefaultSize$ genotypes are assembled and evaluated on all test cases. The genotypes are scored based on the proportion of test cases their associated phenotypes were able to correctly compute. If one a phenotype was able to compute all test cases, then the corresponding genotype is the solution, which is output as the result of the run which terminates successfully , otherwise, go to step 7

7. A negative selection (see 6.9.1) operation which removes genotypes, alleles and possibly whole species from the ecosystem is applied. The ecosystem is then rebuilt to capacity, go to step 6 and proceed to next generation.

One of the features that distinguishes the Abstraction-Based Genetic Programming System from other GP systems is that while evaluation is done at the phenotype level, selection is applied at several level. In ABGP, genotypes are used as statistical tools to evaluate the alleles that compose them and the species upon which pattern they are built. As usual, the fitness of a genotype is a proportionality of how close the genotype's associated phenotype describes a solution, however, the selection of genes and species is implicit as it piggybacks on the selection process applied to genotypes. In ABGP, the first selection procedure occurs at the beginning of the creation of a new generation. This section describes the procedure that creates a new generation of genotypes from an existing generation containing $ecosystemDefaultSize$ evaluated genotypes. This is the iterative step of the GP system and it is applied to a system that evolves an ecosystem eco. The pseudo-code for this step is provided in Program 6.9.1.

Program 6.9.1 The $next\_generation$ function

```
|pre condition 1: alive_eco eco =  \populationDefaultSize |
|pre condition 2: eco.generationCounter = n               |
-----------------------------------------------------------
next_generation eco =
```

Comment 6.9.1 Step 1, see $selection\_filter$, program 6.9.2

```
    selection_filter eco;
```

Comment 6.9.2 Step 2, reconstruction of ecosystem, see section 6.9.2

```
    reconstruct_eco eco
-----------------------------------------------------------
| post condition: alive_eco eco =  \populationDefaultSize |
| post condition: eco.generationCounter = n + 1           |
```

6.9.1 Filter Selection Step

Program 6.9.2 provides the selection procedure pseudo-code. The first step in the production of a new generation from an existing generation is to remove some genotypes from the ecosystem. This step is inspired from the following two effects observed in the biological world:

1. Birth is one of the most dangerous time in the lifetime of an individual and a newborn needs to possess at least a minimum level of fitness in order to reach childhood.
2. Some individuals do not reproduce. This is non-deterministically related to their fitness. In ABGP, this results in a round of selection that aims to remove some lower fitness genotype before they get a chance to pass their genes onto the next generation.
3. A species whose organisms have all died becomes extinct.
4. If all the organisms that carry a given gene in an ecosystem die before reproduction, that gene is removed from the gene pool.

Program 6.9.2 The $selection\_filter$ function

```
| pre condition: alive_eco eco =  \populationDefaultSize                          |
-----------------------------------
selection_filter eco =
    selection_filter_genotypes eco;
    selection_filter_species eco;
    selection_filter_genes eco

-----------------------------------
| post condition: alive_eco eco <=  \populationDefaultSize - \reapPopulationSize |
| post condition: For all (species, population) pairs in eco.kingdom,            |
|    (alive_population population) > 0                                           |
| post condition: For all genes g in eco.genes, g.useInEcosystem > 0             |
```

6.9.1.1 Filter Selection on Genotypes

In ABGP, the initial selection filter step on genotype is composed of two sub-steps. Given an ecosystem eco, the first sub-step is to non-deterministically kill all genotypes in eco with fitness below a certain threshold. "Killing" in this case means the change of status of the genotype from 'Alive' to 'Dead'. This step is deterministic, so the threshold is quite low. After this has been done, the second sub-step compares the number of genotypes marked as 'Alive', $(alive\_eco\ eco)$ to the value $ecosystemDefaultSize - reapPopulationSize$ (where $reapPopulationSize$ is a user supplied parameter that fixes the strength of the selection pressure applied to the population). If the number of genotypes alive is greater than this value, genotypes marked as 'Alive' are killed non-deterministically in proportion to their fitness values until:

$$alive\_eco\ eco \leq ecosystemDefaultSize - reapPopulationSize$$

6.9.1.2 Filter Selection on Species

Similar to the filter selection step on genes (see 6.9.1.3 below), the filter selection step on species is simply a by-product of the selection step on genotypes. Following 6.9.1.1, some species may no longer contain any genotypes marked as 'Alive'. These species are removed (and are referred to as *extinct*). A species that becomes extinct will no longer produce descendent species at the crossover process. At the end of this sub-step, the following predicate holds on eco:

$$\forall(species, population) \in eco.kingdom, (alive\_population\ population) > 0$$

It is at the end of this step that a fitness value is assigned to the species that are still in the ecosystem. This species specific fitness value is computed as :

$$\frac{\text{sum of the fitness values of all genotypes marked as 'Alive' in } population}{(alive\_population\ population)} \tag{6.20}$$

where $population$ is the $Population$ object associated with the species. The fitness of the species is stored in the $fitness$ field of $population$.

6.9.1.3 Filter Selection on Genes

Following 6.9.1.1, some alleles may no longer be carried by any genotypes. These are removed from the gene pool, eventually making space for new alleles. At the end of this sub-step, all the alleles in the gene pool are carried by at least one genotype, and the following predicate holds:

$$\forall\ g \in eco.genes, g.useInEcosystem > 0$$

The number of genotypes that carry an allele g is stored in g's $useInEcosystem$ field and g's $geneFitness$ value can be computed as:

$$\frac{\text{sum of the fitness values of all genotypes marked as 'Alive' in } eco \text{ that carry } g}{g.useInEcosystem}$$

The denominator is necessary because what is calulated here is *the average fitness of a particular allele*. To obtain this value, it is necessary to sum the fitnesses of all the programs that carry the allele and to then divide by the number of such programs.

6.9.2 Reconstruction of Ecosystem Step

The last step of the reconstruction process is the production of new genotypes. This step is virtual, as no new live genotypes are produced. Instead, some new unborn genotypes are produced. These may already exist, or they may be constructed during the recombination process. New alleles and new species may also be produced during the recombination operations, resulting in further increase in the number of 'Unborn' genotypes that live within the system. Once a sufficient amount of unborn genotypes has been created, new genotypes are marked as alive using the diversity optimizing method described in section 6.6.2. The function simply calls the $addNewLiveGenotypes$ function until the predicate $alive\_eco\ eco = ecosystemDefaultSize$ holds. The pseudo-code for $addNewLiveGenotypes$ is given in Program 6.9.4.

Program 6.9.3 The $reconstruct\_eco$ function

```
| pre condition: alive_eco eco <=  \populationDefaultSize - \reapPopulationSize |
| pre condition: For all (species, population) pairs in eco.kingdom,            |
|    (alive_population population) > 0                                          |
| pre condition: For all genes g in eco.genes, g.useInEcosystem > 0             |
-----------------------------------
let reconstruct_eco eco =
    let foundSolution = ref false in
    let sol = ref None in
    while (ecosystemIsNotFull( ) && not(foundSolution)) do
        match addNewLiveGenotypes eco with
            |None --> ( )
            |Some solution -->
            foundSolution := true;
            sol := Some solution
    done;
    eco.generationCounter <- eco.generationCounter + 1

-----------------------------------
| post condition: alive_eco eco =  \populationDefaultSize
```

6.9.2.1 Producing New Genotypes

There are three crossover operations for Abstraction-Based Genetic Programming. Each produces offspring at different genotypic distances from the parents in the search space. The underlying principles of crossover scheme for the system are:

- *Heredity*: offspring should be similar to their parents
- *Variability*: offspring should not be identical to their parents

We applied these principles to both the structure (the species) and the composition (the genes) of the genotype. In ABGP, crossover happens between two parents that belong to the same species. The most simple case of crossover, mutation-free crossover is described below.

6.9.2.2 Crossover Simple Case

The next step is to apply simple crossover. Simple crossover is applied to two living genotypes that belong to the same species. It doesn't introduce any new genetic material in the gene pool and doesn't create any new species. Simple crossover is a basic recombination of the genetic material of the two parents and the offspring produced are somewhere in between both parents' positions in the search space. Each gene carried by the offspring is carried by one of its parents in the same position. An offspring only contains genetic material already carried by one or both of its parents, with the added restriction that it may not be similar to any genotype that already exists in the population (alive or dead). Figure 6.9 illustrates the simple case. In

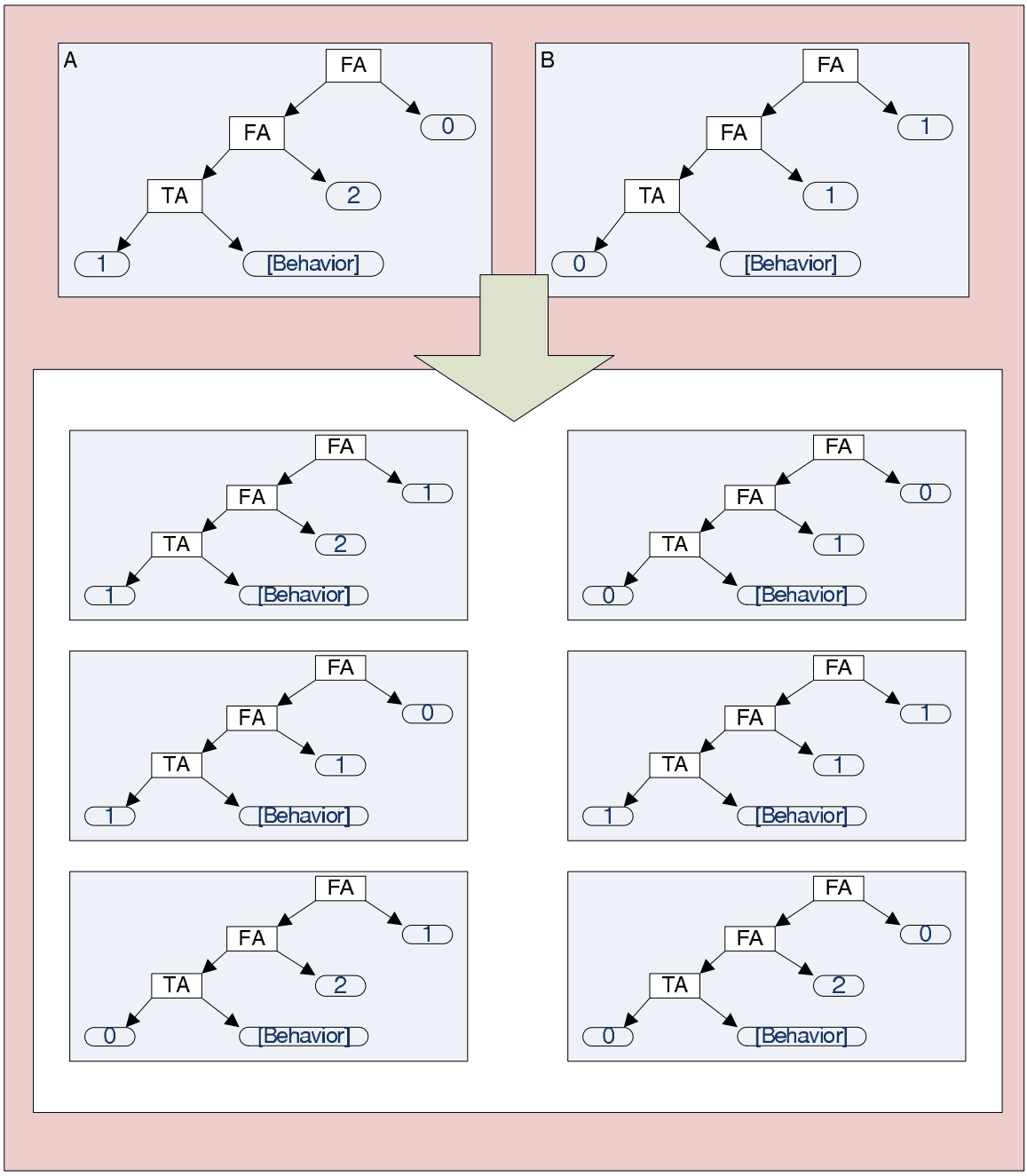

Figure 6.9: The basic crossover case. Any of the 6 possible offspring belongs to the same species that both their parents belong to and each of the offspring genes originates from one or the other parent

Program 6.9.4 The $addNewLiveGenotypes$ function

```
| pre condition: alive_eco eco <=  \populationDefaultSize - \reapPopulationSize |
| pre condition: For all (species, population) pairs in eco.kingdom,            |
|    (alive_population population) > 0
| pre condition: For all genes g in eco.genes, g.useInEcosystem > 0             |
-------------------------------------------------------------------------------

 addNewLiveGenotypes eco  =
    let aliveNum = ref (liveGenotypes( )) in
    let doCrossover sol crossoverOp =
        match sol, liveGenotypes( ) with
            |Some _,_ --> sol
            |None, x when x  > !aliveNum --> None
            |_ --> crossoverOp eco in
    fold_left doCrossover None
    [simpleXover; geneMutationXover; speciesMutationXover]
-------------------------------------------------------------------------------
| post condition: alive_eco eco =  \populationDefaultSize                       |
```

this case, as the pattern on which the parent genotypes are constructed admits 3 slots for gene plugging, there is a total of $2^3 - 2$ possible offspring that may be produced. In general, given two genotypes belonging to the same species with n slots for gene plugging, $2^{n-k} - 2$ offspring may be produced. k is the number of slots in the parents genome that contain similar genes. In figure 6.9, both genotypes belong to the species defined in the upper left portion of figure 6.6.

6.9.2.3 Crossover with Mutation I: Creation of New Genes

Following this step, the value $(alive\_eco\ eco)$ may still (and if it is isn't, eventually will as all the simple recombination possibilities are exhausted) be lower than $ecosystemDefaultSize - reapPopulationSize$. If this is the case, crossover with mutation is applied. This step begins by rebuilding the pool to $maxComp$ using the genes that are in it as seed for new genes. The fitness of a gene is calculated as the average of the fitness of the genotypes that carry it, and it is used to compute the probability function that will be used to select the genes that will undergo the mutation process that will form new genes. New gene are then used to generate new genotypes at crossover time by being inserted in appropriate position (as determined by its type) instead of a parent gene as in the simple case. Figure 6.10 illustrates this subcase of crossover. While the crossover case with gene mutation represent a wider search space exploratory jump than the simple crossover case (a new simple gene might radically change the computational path taken by the program), it only produces offspring within the search space partition that is defined by the species to which the offspring's genotype belong. The next crossover scheme is more radical as it produces offspring outside the search space partition defined by the parent's species.

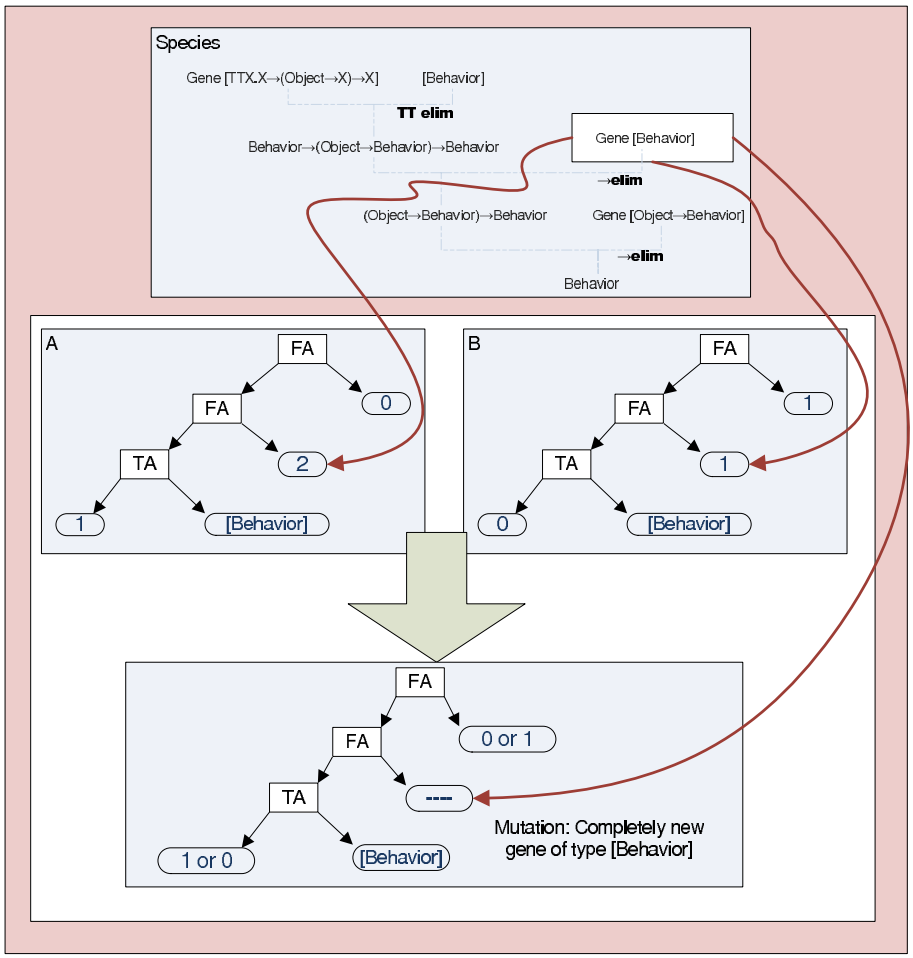

Figure 6.10: Crossover with mutation generated by the introduction of a new gene in the gene pool.

6.9.2.4 Crossover with Mutation II: Creation of New Species

The last mutation case is the one that creates a new species. As in nature, the first genotypes of a new species must be the offspring of genotypes that belong to a different species. In this crossover scenario, it is the species to which the two parent genotypes belong that is mutated. A new descendent species is produced from the ancestor species to which the genotypes belong. This is a macro mutation scenario and large structural modifications have a lower probability of producing high quality genotypes because they represent a wider jump in the search space than either simple crossover or crossover with gene mutation. In order to compensate, when this crossover scheme is applied, all possible combinations deriving from the ancestor species are produced, sampling a wider subset of the search space partition defined by the species. Figure 6.11 illustrates this crossover scenario. This crossover scheme is applied a maximum of two times in a given generation and only in one of the following situations:

- The two genotypes chosen for reproduction both belong to the highest 10th percentile of the population in terms of fitness.
- All possible offspring (including those produced by gene mutation crossover) have already been eval-

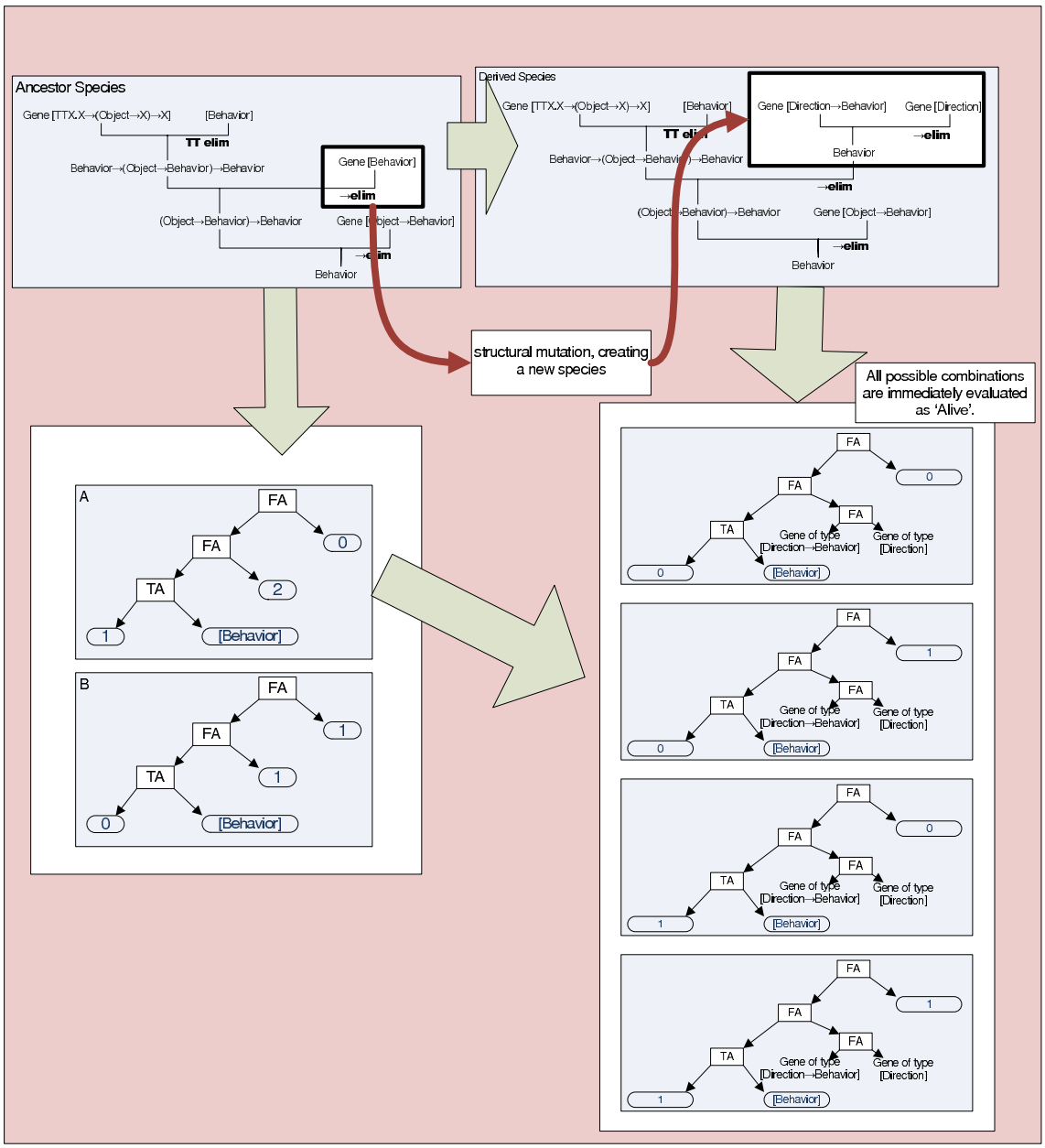

Figure 6.11: In this crossover scenario, a new species is created from the species to which both parent belong. The gene slots that match in the offspring species and in the parent species are filled as in the first case. The gene slots that do not match are filled from random choices in the gene pool

uated for the two genotypes selected for reproduction.

- There hasn't been a "better than ever before" genotype produced in the last 3 generations (as a stagnation preventing measure)

6.10 CONCLUSION

In this chapter, we presented the ABGP system. We've built an implementation of this system and in the next chapter, we describe the results that we've been able to obtain with it.

Chapter 7

Results

This section lists a series of experiments on which the capabilities of the system were tested. In all the experiments described in this chapter, the selection filter process removes a third of the genotypes alive at the beginning of each generation. Other parameters are set individually for each experiment, and we provide the user supplied values at the beginning of the experiment's description in the following format:

```
maxGeneComp = 55
maxComp = 5000
geneDifference = 12
populationDefaultSize = 200
```

Where *maxGeneComp* indicates the maximum complexity size of an allele, *maxComp* is the maximum total complexity size of the gene pool, *geneDifference* is the value that measures the "difference" factor that must hold in between alleles and $ecosystemDefaultSize$ is the number of genotypes that are evaluated (marked as 'Alive') in each generation.

7.1 Boolean Functions

A substantial portion of our experimental capital was spent on the study of the evolution of boolean argument functions. There are two reasons for this:

1. Boolean formulas can be used to compute quite complicated functions, and it is natural to consider their learnability

2. Our first motivator for using System F as an underlying representation scheme was its expressive power. In this case, we demonstrate the advantage of the system as a GP language by evolving all functions in this section without defining any terminals that need to be evaluated outside the system. Instead, we used a terminal set exclusively composed of names for System F pure abstract terms and types (expressions that do not contain constants). We've already described the representation of Boolean functions as pure abstraction in section 5.4.3.1. In this section, we make use of this encoding for all the problems described.

All results in this section were obtained using context BooleanDef (7.1.1)

context 7.1.1 *BooleanDef*

```
/*Unique type defined as pure abstraction */
TyBool = TT X . X -> X ->X;

/*The two type constructors                */
true = Lam X . lam x : X . lam y : X . x;
false = Lam X . lam x : X . lam y : X . y;
```

7.1.1 The *not* Boolean Function

In this problem, we evolve a program that evolves a one argument *not* function using the following list of test cases:

```
Arg (TyBool),Result (TyBool)
true, false
false, true
```

when using the following set of initial variables:

```
maxGeneComp = 50
maxComp = 1000
geneDifference = 12
populationDefaultSize = 10
```

and performing 100 runs:

- 42% of the runs discover a solution on the first generation
- 23% of the runs discover a solution on the second generation
- 17% of the runs discover a solution on the third generation
- 18% of the runs discover a solution on the fourth generation

Each generation after the first one destroys 3 live genotypes and evaluates 3 new genotypes, so by the fourth generation a maximum of 19 genotypes have been evaluated with a 100% chance of finding a solution. However, simply running a first generation (10 evaluations) twice for a total of 20 generations only produces an 84% chance of finding a solution. This indicates that the discovery process is not random. The maximum fitness score is 2 (as there are two test cases). The average fitness on:

- Generation 1 is 0.39
- Generation 2 is 0.625
- Generation 3 is 0.84
- Generation 4 is 0.98

Which demonstrates that there is an increase of fitness with the passing of generations.

7.1.1.1 Solution example

As mentioned, the system always finds a solution (within 4 generations). One of the evolved one allele-species solutions was:

```
(/*  Gene = (TT A. A-> A-> A)-> (TT O. O-> O-> O)-- ID = 5*/
     lam x:TT A. A-> A-> A.x [TT K. K-> K-> K]
        (Lam O.lam m:O.lam s:O.s)
        (Lam O.lam c:O.lam r:O.c))
```

which normalizes to:

$$\lambda e^{TyBool}.e\ [TyBool]\ (false)\ (true) \tag{7.1}$$

The program belongs to the species illustrated by figure 7.1.

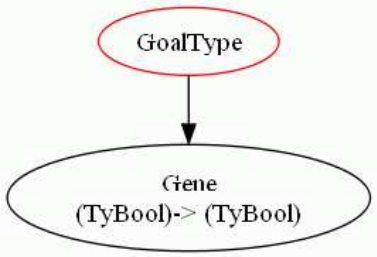

Figure 7.1: Species to which the example program (not function) that normalizes to 7.1 belongs

7.1.2 The and Boolean Function

For this problem, the system evolves a two argument and function using the following list of test cases:

```
Arg1(TyBool),Arg2(TyBool),Result(TyBool)
true, true, true
false, true, false
true, false, false
false, false, false
```

Using the same variables:

```
maxGeneComp = 50
maxComp = 1000
geneDifference = 12
populationDefaultSize = 10
```

The system finds a solution (over 100 runs):

- 8% of the time on the 1st generation
- 11% of the time on the 2nd generation
- 33% of the time on the 3rd generation

- 36% of the time on the 4th generation
- 12% of the time on the 5th generation

7.1.2.1 Solution example

One of the evolved solutions was:

```
(/* Gene = ((TT W. W-> W-> W)-> (TT I. I-> I-> I)->
                (TT F. F-> F-> F)-> (TT E. E-> E-> E))->
                (TT J. J-> J-> J)-> (TT J. J-> J-> J)->
                (TT U.U-> U-> U)-- ID = 4*/
    lam a:(TT I. I-> I-> I)-> (TT P. P-> P-> P)->
          (TT K. K-> K-> K)-> (TT T. T-> T-> T).
    lam w:TT M. M-> M-> M.a w w)

(/* Gene = (TT K. K-> K-> K)-> (TT J. J-> J-> J)->
             (TT L. L-> L-> L)-> (TT I. I-> I-> I)-- ID = 5*/
    lam w:TT Y. Y-> Y-> Y.
        w [TT U. U-> U-> U]
            (Lam N.lam k:N.lam q:N.q)
            (Lam F.lam k:F.lam j:F.k) [TT E. E-> E-> E])
```

which simplifies to:

$$\lambda i^{TyBool}.i\ [TyBool]\ false\ true\ [TyBool]\ i \tag{7.2}$$

The program belongs to the species illustrated by figure 7.2

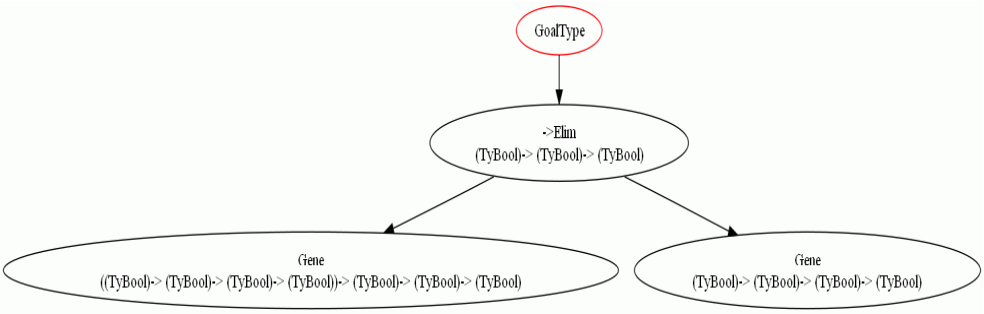

Figure 7.2: Species to which the example program (and function) that normalizes to 7.2 belongs

7.1.3 The *or* Boolean Function

For this problem, our system evolves a two argument *or* function using the following list of test cases:

```
Arg1(TyBool),Arg2(TyBool),Result(TyBool)
true, true, true
false, true, true
true, false, true
false, false, false
```

Using the set of variables:

```
maxGeneComp = 50
maxComp = 1000
geneDifference = 12
populationDefaultSize = 10
```

The system finds a solution (over 100 runs):

- 3% of the time on the 1st generation
- 15% of the time on the 2nd generation
- 24% of the time on the 3rd generation
- 33% of the time on the 4th generation
- 13% of the time on the 5th generation
- 12% of the time on the 6th generation

7.1.3.1 Solution example

The system always finds a solution. For example:

```
(/* Gene =  (TT R. R-> R)-> (TT V. V-> V-> V)->
            (TT J. J-> J-> J)-> (TT V. V-> V-> V)--ID = 18*/
    lam q:TT S. S-> S.
    lam v:TT P. P-> P-> P.v [TT K. K-> K-> K] v)

(/* Gene = TT G. G-> G-- ID = 0*/
    Lam I.lam k:I.k)
```

which simplifies to the ingenuous solution:

$$\lambda i^{TyBool}.i\ [TyBool]\ i \tag{7.3}$$

Compare, for example with the more obvious but longer:

$$\lambda o^{TyBool}.\lambda t^{TyBool}.o\ [TyBool]\ (true)\ t \tag{7.4}$$

The solution program belongs to the species illustrated by figure 7.3 (this result was obtained on an older version of the system. The symbol 'ProgNode' should read 'Gene' and 'ImplyElim' should read '→elim').

7.1.4 The xor Boolean Function

The preceding functions were relatively easier to evolve than an xor. To evolve xor, we use:

```
maxGeneComp = 50
maxComp = 5000
geneDifference = 12
populationDefaultSize = 100
```

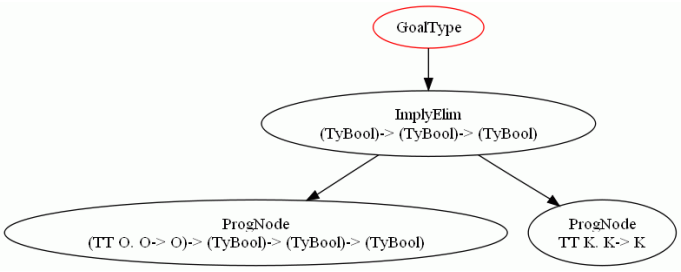

Figure 7.3: Species to which the example program (*or* function) that normalizes to 7.3 belongs

Over a set of 100 runs, we obtain a solution in

- 2% of the time on the 1st generation
- 10% of the time on the 2nd generation
- 24% of the time on the 3rd generation
- 23% of the time on the 4th generation
- 17% of the time on the 5th generation
- 13% of the time on the 6th generation
- 11% of the time on the 7th generation

7.1.4.1 Solution example

The system always find a solution (but with more effort):

```
(((/* Gene = (TT Y. Y-> Y-> Y)->
              (TT M. (M-> M)-> (TT D. D-> D-> D)-> M-> M)->
              ((TT O. O-> O-> O)-> (TT R. R-> R-> R))->
              (TT T. T-> T-> T)->
              (TT S. S-> S-> S)->
              (TT P. P-> P-> P)-- ID = 0*/
    lam q:TT Y. Y-> Y-> Y.
    lam d:TT I. (I-> I)-> (TT V. V-> V-> V)-> I-> I.d [TT O. O-> O-> O])

(/* Gene = TT P. P-> P-> P-- ID = 1*/
    Lam L.lam k:L.lam n:L.k))

(/* Gene = TT N. (N-> N)-> (TT U. U-> U-> U)-> N-> N-- ID = 0*/
    Lam K.lam n:K->K.lam h:TT N. N-> N-> N.lam c:K.h [K] (n c) c))

(/* Gene = (TT Y. Y-> Y-> Y)-> (TT J. J-> J-> J)-- ID = 4*/
    lam r:TT D. D-> D-> D.
        r [TT D. D->D-> D]
            (Lam K.lam s:K.lam w:K.w)
```

```
(Lam S.lam n:S.lam x:S.n))
```

which simplifies to the term:

$$\lambda j^{TyBool}.\lambda p^{TyBool}.j\ [TyBool]\ (p\ [TyBool]\ (false)\ (true))\ p \tag{7.5}$$

The program belongs to the species illustrated by figure 7.4

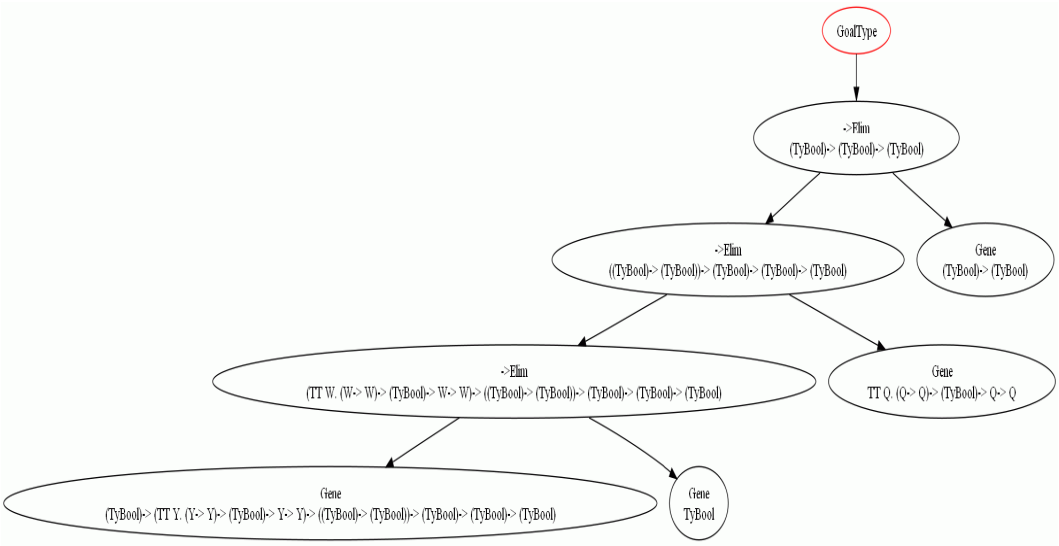

Figure 7.4: Species to which the example program (xor function) that normalizes to 7.5 belongs

7.1.5 The Half-adder Boolean Function

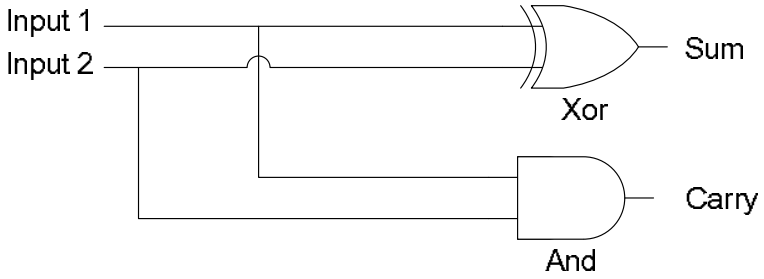

Figure 7.5: A half-adder circuit

In this problem, we evolve a two argument $half\text{-}adder$ function. The half-adder circuit produces two output: The carry and the sum resulting from the addition of the two 1-bit arguments. As an electronic circuit, it might be built as per schema of figure 7.5. we use:

```
maxGeneComp = 50
maxComp = 5000
geneDifference = 12
populationDefaultSize = 1000
```

We evolved the function using the following list of test cases:

```
Arg1(TyBool),Arg2(TyBool),Result(TT X .TyBool->TyBool->X)
true,true, Lam X . lam x : TyBool->TyBool->X.x true false
false, true, Lam X . lam x : TyBool->TyBool->X.x false true
true, false, Lam X . lam x : TyBool->TyBool->X.x false true
false, false, Lam X .lam x : TyBool->TyBool->X.x false false
```

in which the function's output is encoded as a pair (the first element is the carry, the second is the sum). We give no indication to the system as to what the structure of a pair is, or how to form one, beyond what is encoded in the test cases. The context is sill completely defined by context 7.1.1. We get a solution within 9 generations 89% of the time.

7.1.5.1 Solution example

An example solution:

```
(/* Gene = ((TT V. V-> V-> V)-> (TT O. O-> O-> O)->
    (TT O. O-> O-> O))-> (TT R. R-> R-> R)->
    (TT W. W-> W-> W)-> (TT V. (TyBool-> TyBool-> V)-> V)-- ID = 0*/

    lam m:TyBool-> TyBool-> TyBool.
    lam y:TyBool.lam j:TyBool.
    Lam M.lam u:TyBool-> TyBool-> M.u (y [TT P. P-> P-> P] j y) (m y j))

((/* Gene = ((TT Y. Y-> Y-> Y)-> (TT Y. Y-> Y-> Y))->
            (TT V. V-> V-> V)-> (TT N. N-> N-> N)->
            (TT A. A-> A-> A)-- ID = 0*/

    lam r:(TT X. X->X-> X)-> (TT G. G-> G-> G).
    lam n:TT R. R-> R-> R.
    lam v:TT G. G-> G-> G.n [TT R. R-> R-> R] (r v) v)

(/* Gene = (TT J. J-> J-> J)-> (TT Q. Q-> Q-> Q)-- ID = 18*/

    lam r:TyBool.r [TT U. U-> U-> U]
        (Lam X.lam o:X.lam t:X.t)
        (Lam L.lam c:L.lam h:L.c)))
```

Corresponding to the normalization:

$$\begin{aligned}&\lambda y^{TyBool}.\lambda e^{TyBool}.\\&\qquad \Pi M.\lambda x^{TyBool\to TyBool\to M}.\\&\qquad x\ (y\ [TyBool]\ e\ y)\ (y\ [TyBool]\ (e\ [TyBool]\ (false)\ (true))\ e)\end{aligned} \tag{7.6}$$

The program belongs to the species illustrated by figure 7.6.

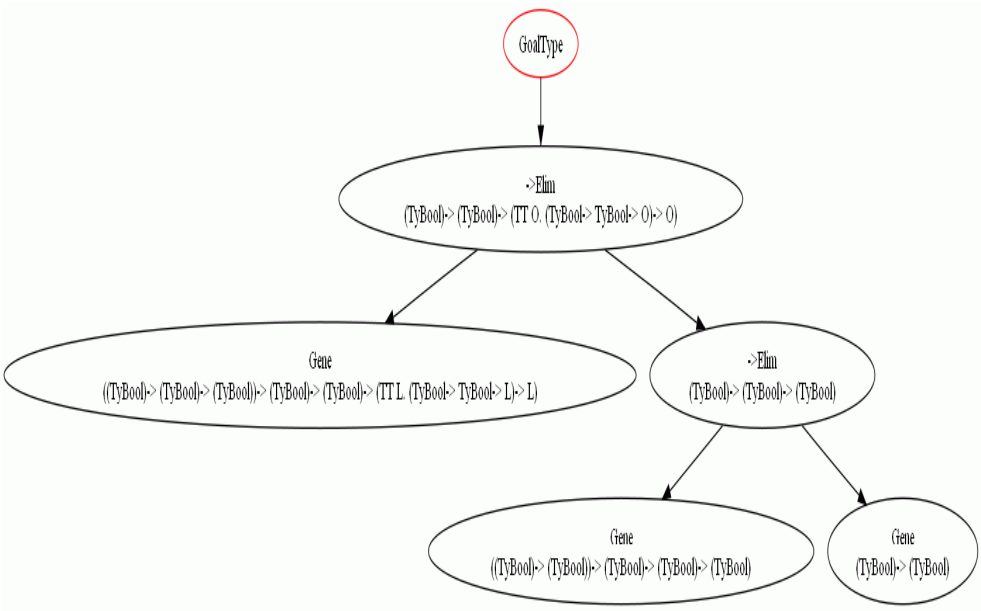

Figure 7.6: Species to which the example program (half-adder) that normalizes to 7.6 belongs

7.2 Lists and Lists Operations

We experimented with lists of integers using the type:

$$TyList \stackrel{\text{def}}{=} \Pi X.X \to (Int \to X \to X) \to X \tag{7.7}$$

All results in this section were obtained using context 7.2.1: The ABGP system is able to consistently evolve a solution to the following problems within 3000 genotype evaluations with 99% probability (computed out of 150 runs for each problems). For each of these problems, the parameters were fixed at $maxComp$= 5000 units, $ecosystemDefaultSize$= 3000, $maxGeneComp$=45 and $geneDifference$=15. All the solutions evolved work for arbitrary list size and the exact same system was used to evolve all of the solutions. The only difference between the runs were the lists of test cases passed as input.

7.2.1 Summing the Elements of a List of Numbers

In this problem, we evolve a program that adds recursively the elements of a list of numbers, returning 0 if the list is empty. In this case, the ABGPS is always able to always find a solution to this problem within 300 evaluations.

context 7.2.1 $ListInt$

```
/*Types*/
TyList = TT X . X -> (TyInt->X->X)->X;

/*List constructors*/
empty_list = Lam X . lam x : X . lam y : TyInt->X->X . x;
cons =  lam elt : TyInt .
        lam lst : TyList .
        Lam X . lam x : X . lam y : TyInt->X->X . y elt (lst [X] x y);

/*Operations$
plus : TyInt->TyInt->TyInt;
mult : TyInt->TyInt->TyInt;
minus : TyInt->TyInt->TyInt;
div : TyInt->TyInt->TyInt;

/*First order objects*/
zero : TyInt;
one : TyInt;
ten : TyInt;
```

7.2.1.1 Solution example

Example of solution program:

```
(/* Type = TyInt-> (TT E. E-> (TyInt-> E-> E)-> E)-> TyInt --
    ID = 3*/ lam n:TyInt.lam r:TyList.r [TyInt] n plus)
((/* Type = (TT W. W-> (TyInt-> W-> W)-> W)-> TyInt--
    ID = 17*/ lam o:TyList.o [TyInt] 0 mult)
(/* Type = TT C. C-> (TyInt-> C-> C)-> C--
    ID = 0*/Lam D.lam j:D.lam u:TyInt-> D-> D.j))
```

Corresponding to the normalized term:

$$\lambda h^{TyList}.h\ [TyInt]\ zero\ plus \tag{7.8}$$

This is a short solution, but the reader must keep in mind that this is the *normalized term corresponding to the solution*. The solution that the GP produced is in fact longer but gets compacted when normalized. The solution in this case belongs to the species depicted in figure 7.7.

7.2.2 Multiplying the Elements of a List of Numbers

Our system is able to evolve a program that recursively multiplies all the elements of a list of numbers and returns 1 if the list is empty. In this case as in the others, the size of the list is arbitrary. To evolve this program, we use the list of 7 test cases below:

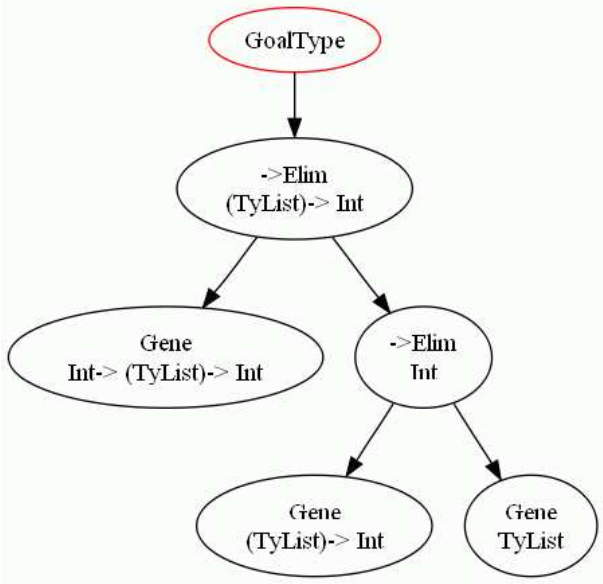

Figure 7.7: Species to which the example program that normalizes to 7.8 belongs

```
Arg (List of Int),Result
cons 0 (cons 6 empty_list),0
cons 4 (cons 6 empty_list),24
cons 2 (cons 2 (cons 2 empty_list)),8
cons 2 (cons 5 (cons 2 empty_list)),20
```

7.2.2.1 Solution example

Example of solution program:

```
(/*
Type = (((TT W. W-> (Int-> W-> W)-> W)-> Int)->
(TT N. N-> (Int-> N-> N)-> N)-> Int)->
(TT V. V-> (Int-> V-> V)-> V)-> Int-- ID = 0
*/
lam s:((TT H. H-> (Int-> H-> H)-> H)-> Int)-> (TT Q. Q-> (Int->
Q-> Q)-> Q)-> Int.s (lam t:TyList.t [Int] 1 mult))

(/*
Type = ((TT T. T-> (Int-> T-> T)-> T)-> Int)->
(TT H. H-> (Int-> H-> H)-> H)-> Int-- ID = 0*/
lam p:(TT S. S-> (Int-> S-> S)-> S)-> Int.p)
```

Corresponding to the term:

$$\lambda n^{TyList}.n\ [Int]\ one\ mult \tag{7.9}$$

This program belongs to the species illustrated by figure 7.8

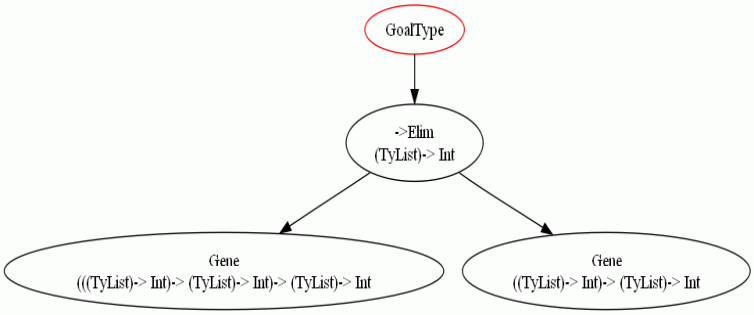

Figure 7.8: Species to which the solution listed for the parity of a list problem belongs

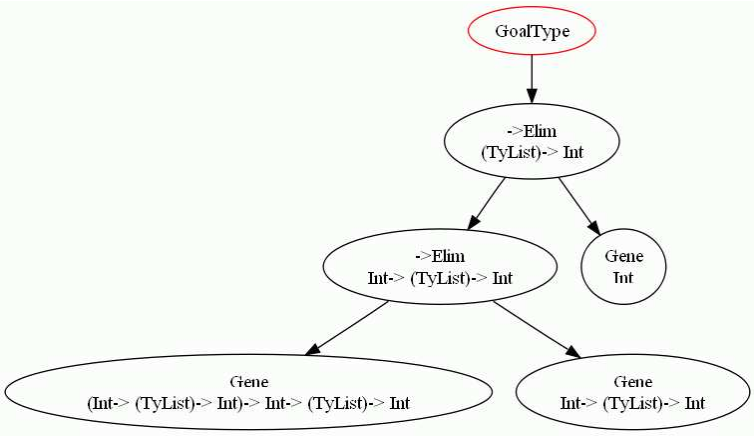

Figure 7.9: Species to which the example program that normalizes to 7.10 belongs

7.2.3 Finding the Length of a List of Numbers

We are able to evolve a program that returns the number of elements of a list of numbers (0 if empty) passed as its argument within 10 generations using these parameters:

```
maxGeneComp = 55
maxComp = 5000
geneDifference = 12
populationDefaultSize = 1000
```

7.2.3.1 Solution example

Below is an example solution:

$$\lambda d^{TyList}.d\ [Int]\ zero\ (\lambda j^{Int}.\lambda k^{Int}.plus\ k\ 1) \tag{7.10}$$

7.2.4 Parity of List Size

We are able to always evolve, within 15 generations, a program that takes a list of numbers as its only argument and returns 1 if the size of the list is even and 0 otherwise (the program returns 1 if the list is only

composed of the $empty\_list$). To evolve this program, we use the following list of test cases:

```
Arg (List of Int), Result
cons 6 empty_list,0
cons 4 (cons 6 empty_list),1
cons 2 (cons 2 (cons 2 empty_list)),0
empty_list,1
cons 89 (cons 2 (cons 0 empty_list)),0
cons 0 (cons 2 (cons 0 empty_list)),0
cons 3 (cons 0 (cons 2 (cons 0 empty_list))), 1
```

with the set-up:

```
maxGeneComp = 55
maxComp = 5000
geneDifference = 12
populationDefaultSize = 1000
```

7.2.4.1 Solution example

Example of solution program:

```
(/* Type = Int-> (TT T. T-> (Int-> T-> T)-> T)-> Int--
    ID = 1*/ lam n:Int.lam l:TyList.l [Int] n (lam x:Int.lam u:Int.minus 1 u))
((/* Type = Int-> Int--
    ID = 4*/ lam f:Int.f)
(/* Type = Int--
    ID = 0*/ one))
```

Corresponding to the term:

$$\lambda x^{TyList}.x\ [Int]\ 1\ (\lambda w^{Int}.\lambda y^{Int}.minus\ 1\ y) \tag{7.11}$$

This phenotype belongs to the species depicted in figure 7.10

7.3 TREES AND TREE OPERATIONS

We experimented with binary trees of integer as represented by the type:

$$TyTree \stackrel{\text{def}}{=} \Pi X.(Int \to Int \to Int \to X) \to (Int \to X \to X \to X) \to X \tag{7.12}$$

All results in this section were obtained using context 7.3.1.
For example, 7.13 is the binary tree of type $[TyTree]$ that normalizes to 7.14.

$$stem\ 0\ (branch\ 7\ 6\ 8)\ (branch\ 3\ 1\ 4) \tag{7.13}$$

$$\Lambda X.\lambda x^{Int\to Int\to Int\to X}.\lambda y^{Int\to X\to X\to X}.y\ 0\ (x\ 7\ 6\ 8)\ (x\ 3\ 1\ 4) \tag{7.14}$$

An abstract representation of this tree is depicted in figure 7.11

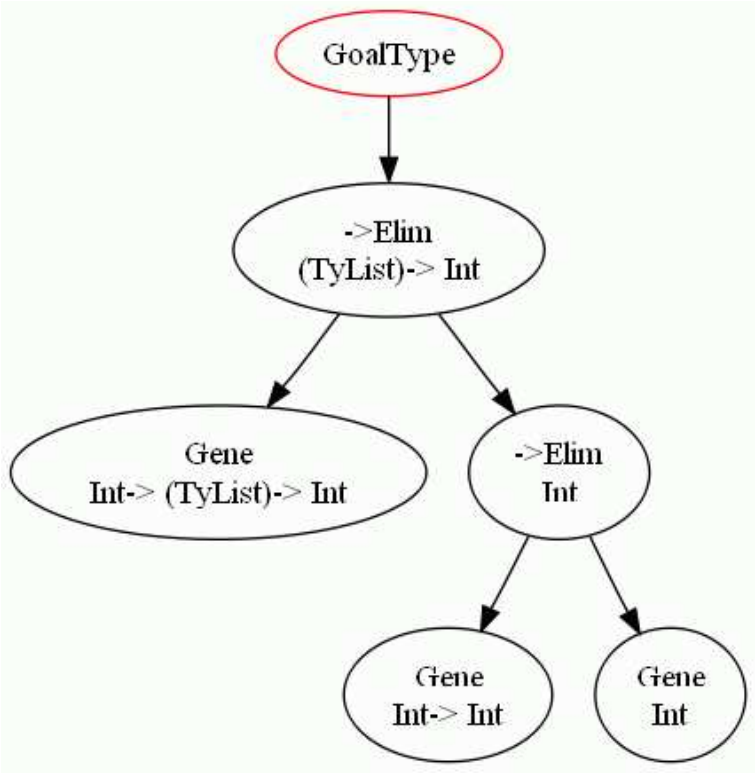

Figure 7.10: Species to which the solution for the list parity problem belongs (term 7.11)

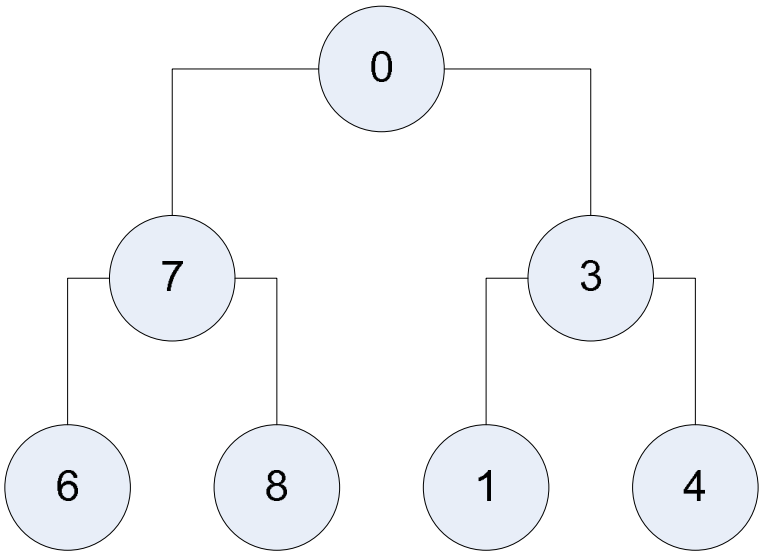

Figure 7.11: The object of type $[TyTree]$ corresponding to 7.13

context 7.3.1 The tree context

```
/*Types*/
TyBool = TT X . X -> X ->X;
TyTree = TT X .(Int->Int->Int->X)->(Int->X->X->X)->X;

/*Boolean terms*/
true  = Lam X . lam x : X . lam y : X . x;
false = Lam X . lam x : X . lam y : X . y;

/*Tree construction terms*/
branch = lam a : Int .
         lam b : Int .
         lam c : Int .
         Lam X . lam x : Int->Int->Int->X .
                 lam y : Int->X->X->X . x a b c;

stem =   lam a : Int .
         lam b : TyTree .
         lam c : TyTree .
         Lam X . lam x : Int->Int->Int->X .
                 lam y : Int->X->X->X . y a (b [X] x y) (c [X] x y);

/*Binary arithmetics*/
plus : Int->Int->Int;
mult : Int->Int->Int;
minus : Int->Int->Int;
div : Int->Int->Int;

/*Comparison*/
gt : Int->Int->TyBool;

/*Numerical Constants*/
zero : Int;
one : Int;
ten : Int;
```

7.3.1 Path that Results in the Maximum Node Sum

In this problem, we evolve a program that discovers the maximum sum path in a tree. For example, in tree 7.14, the maximum sum path is $0+7+8=15$. To evolve this program, we used the list of 5 test cases, listed below:

```
Arg (Tree of Int), Result (Int)
(stem 0 (branch 7 6 8) (branch 3 1 4)), 15
(stem 1 (branch 7 4 8) (stem 3 (branch 14 15 16)(branch 3 1 4))),34
(stem 1 (branch 1 4 8) (stem 3 (branch 9 1 6)(branch 3 1 4))), 19
```

```
(stem 1 (stem 5 (branch 1 4 8)(branch 9 4 8)) (stem 3 (branch 9 1 6)(branch 3 1 4))),23
(stem 0 (stem 5 (branch 1 4 2)(branch 4 4 8)) (stem 3 (branch 2 1 6)(branch 3 1 4))),17
```

Our system consistently finds a solution. Below is an example of a solution program:

```
(/* Type =  (Int-> Int-> Int-> Int)->
            (TT X. (Int-> Int-> Int-> X)-> (Int-> X-> X -> X)-> X)-> Int--
ID = 4*/ lam y:Int-> Int-> Int-> Int.lam j:TyTree.j [Int] y y)
((/* Type = (Int-> Int-> Int)-> Int-> Int-> Int-> Int--
ID = 6*/lam i:Int-> Int-> Int.lam r:Int.lam h:Int.lam h':Int.plus r (i h h'))
(/* Type = Int-> Int-> Int--
ID = 2*/lam h:Int.lam j:Int.gt h j [Int] h j))
```

Corresponding to the term:

$$\begin{aligned}&\lambda x^{TyTree}.x\ [Int]\\&\quad(\lambda k^{Int}.\lambda v^{Int}.\lambda g^{Int}.plus\ k\ (gt\ v\ g\ [Int]\ v\ g))\\&\quad(\lambda w^{Int}.\lambda t^{Int}.\lambda y^{Int}.plus\ w\ (gt\ t\ y\ [Int]\ t\ y)))\end{aligned}\tag{7.15}$$

The solution in this case belongs to the species depicted by figure 7.12.

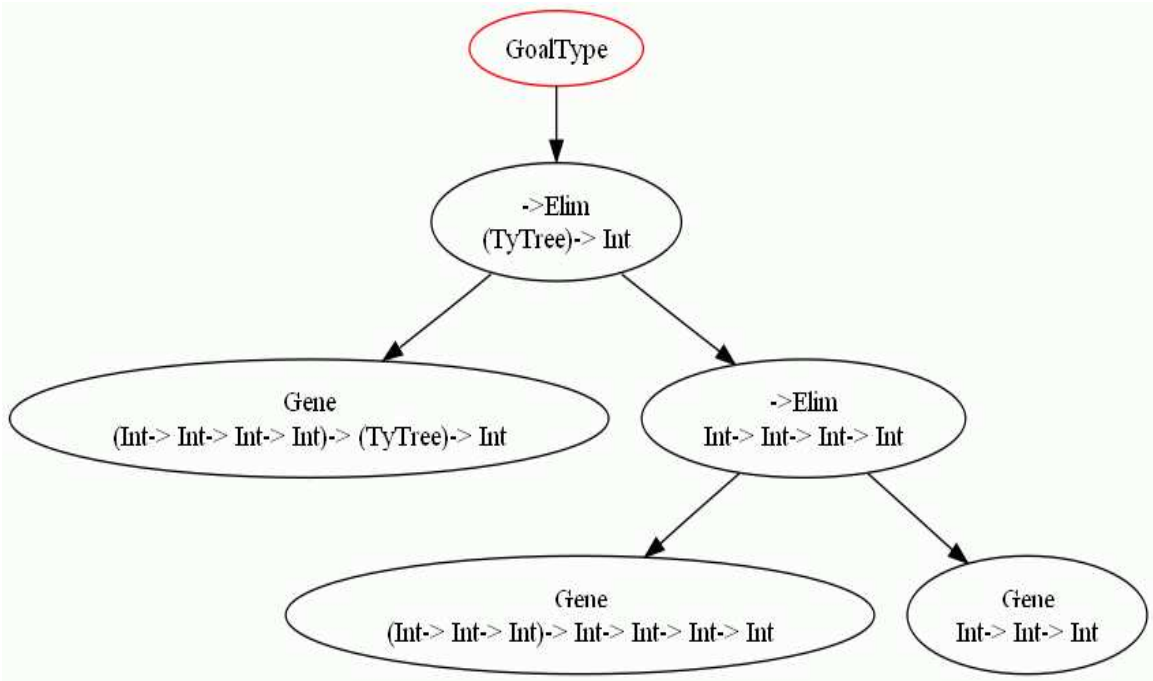

Figure 7.12: The species of the solution program corresponding to the term of 7.15

7.4 STRINGS AND LIST OF CHARACTERS

We experimented with strings (strings were built in the system) and lists of strings using context 7.4.1. so that for example,

context 7.4.1 $ListStr$, a context for evolving list operations

```
/*******************************
Context ListStr for strings and list of strings programs
*******************************/

TyList = TT X . X -> (String->X->X)->X;
TyBool = TT X . X -> X ->X;

true = Lam X . lam x : X . lam y : X . x;
false = Lam X . lam x : X . lam y : X . y;

empty_list = Lam X . lam x : X . lam y : String->X->X . x;
cons = lam elt : String . lam lst : TyList .
    Lam X . lam x : X . lam y : String->X->X . y elt (lst [X] x y);

empty_string = "";

length : String->Int;
concat : String->String->String;
compstr : String -> String -> TyBool;
substr : String -> Int -> Int -> String;
```

```
(cons "dsds" (cons "jk" empty_list))
```

normalizes to:

```
(Lam H.lam o:H.lam k:String-> H-> H.k "dsds" (k "jk" o))
```

We easily evolve a list concatenation operation using only the following 2 test cases:

```
Arg (List of Strings), Result
(cons "hsja" (cons "hsj" (cons "ty" empty_list))), "hsjahsjty"
empty_list,empty_string
```

The programs evolved correspond usually (with some exceptions) to the term:

$$\lambda x^{TyList}.x\ [String]\ empty\_string\ concat \tag{7.16}$$

7.5 THE ANT COOPERATION EVOLUTION SYSTEM

Ant problems are typical in evolutionary computation. We tested ABGP's potential to evolve such systems using the following variables:

```
maxGeneComp = 55
maxComp = 5000
geneDifference = 12
```

```
populationDefaultSize = 200
```

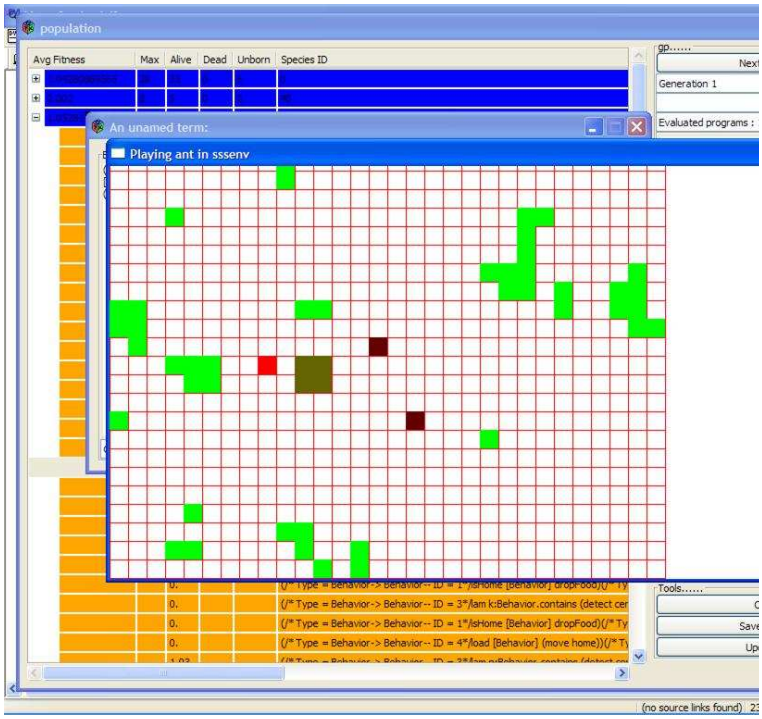

Figure 7.13: The ant cooperation problem interface. The bright red pixel represents an ant carrying food. The other two brown pixels are ants looking for food. The dark green square is the nest and the green patches are the food

In this experiment, 8 ants, guided by the rules of their common genotype are evaluated in a 2-dimensional environment (see figure 7.13) for 200 time units. At the beginning of each evaluation run, the environment contains 60 particles of food. Ants must pick up food and carry it back home. They can "communicate" with each other using pheromones, which they can drop in the environment. The scent that is given by the pheromones decay as time ticks on. Pheromones are implemented as real numbers (between 0 and 1.0). Once a pheromone is placed on the grid, its value is divided by two at each click, and it is removed when its value is below 0.00001. The source code and win32 executable built for this experiment is available online at: `http://code.google.com/p/gene-centered-ecosystem-framework-gp/source/browse/trunk#trunk/doc/franckbinardPhDthesis/experimental/system_ants`. For this experiment, we used context 7.5.1. Genotypes evaluate to programs of type $Behavior$.

7.5.1 Procedure

Each of the 8 ants guided by their genotype behavior is "executed" in sequence for each genotype. The genotype's fitness is computed as the number of food particles that the family managed to bring back home (from 0 to 60) plus a token score (.5) simply for not producing a run time error. Run time errors might be caused, for example, by a call to the *dropFood* function by an ant that is not carrying any food. The following genotype:

context 7.5.1 $AntCtx$

```
/*Types*/
TyBool = TT X . X->X->X;
Direction;
Behavior;

/*Sensing and acting*/
left : Direction;
right : Direction;
front : Direction;
random : Direction;
home : Direction;
move : Direction -> Behavior;
facingHome : TyBool;
onFood : TyBool;
amHome : TyBool;
amLoaded : TyABool;

detectMarker : TT X . (Direction -> X) -> X -> X ;
detectFood : TT X . (Direction -> X) -> X -> X ;

dropFood : Behavior;
pickupFood : Behavior;
dropMarker : Behavior;
goHome : Behavior;
turnAround : Behavior;
```

```
/* Type = Behavior-- ID = 42*/
    detectMarker [Behavior] move
        (amLoaded [Behavior] (amHome [Behavior] dropFood (moveA home))
        (onFood [Behavior] pickupFood (detectFood [Behavior] moveA
        (detectMarker [Behavior] moveA (moveA (facingHome [DirectionA]
        right random))))))
```

is an example of a one allele (type $[Behavior]$) complete program. Its fitness was measured at 26.5 but it would vary in different environments. The next genotype:

```
(/* Type = Behavior-> Behavior-- ID = 1*/
    detectFood [Behavior] (lam s:DirectionA.moveA s))
(((/* Type = (DirectionA-> Behavior)-> Behavior-> Behavior-- ID = 1*/
    detectFood [Behavior])
(/* Type = DirectionA-> Behavior-- ID = 0*/moveA))
```

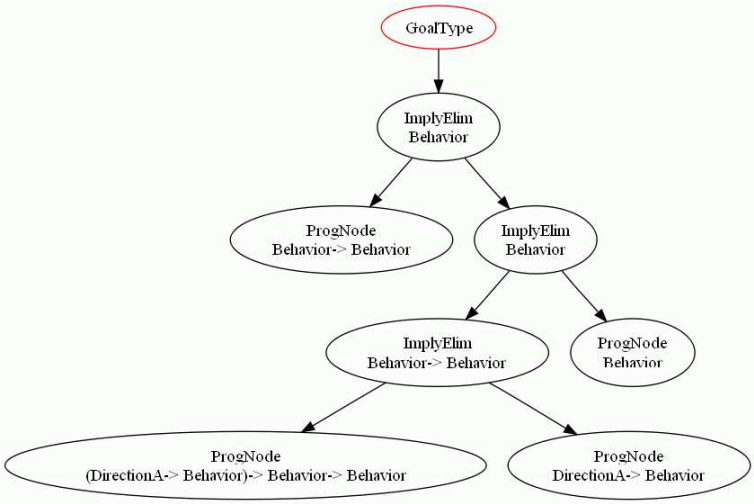

Figure 7.14: Species of genotype example for ant problem

```
(/* Type = Behavior-- ID = 0*/
    amLoaded [Behavior]
    (amHome [Behavior] dropFood (moveA home))
    (onFood [Behavior] pickupFood (detectFood [Behavior] moveA
    (detectMarker [Behavior] moveA (moveA (facingHome [DirectionA] right random))))))
```

is a 4 alleles genotype, that scored 13.5. It belongs to the species depicted in figure 7.14. Figure 7.15 plots the average fitness changes over 10 generations for 50 runs.

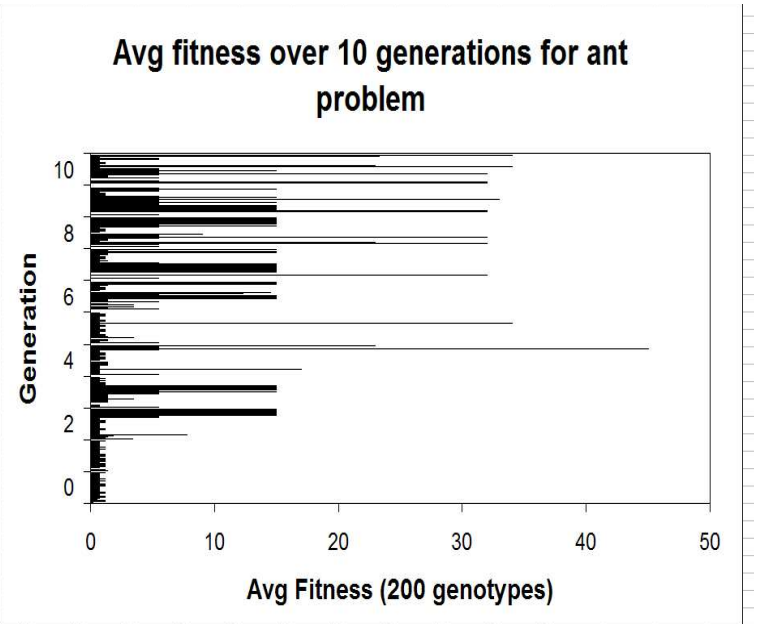

Figure 7.15: Species of genotype example for ant problem

7.6 DISCUSSION

In this section, the results presented above are discussed.

7.6.1 Importance of $maxGeneComp$ Value

Our results indicate a strong relationship between the average size of alleles and their performance. Our data for this set of problems strongly suggests that allowing larger alleles reduces performance when the maximum total size of the gene pool remains constant. This is illustrated by figure 7.16 which describes the results obtained by solving each of the three problems at different values of $maxGeneComp$ 50 times for a maximum of 20 generations. However, when genes above a certain size are not allowed, it becomes impossible to derive genes capable of expressing some computations above a threshold of complexity. When this happens, solutions become again harder to come by as the evolution process needs to compensate by creating species that use many very small genes to express the computation. For the xor function for example, the optimum value seems to be around 45 units.

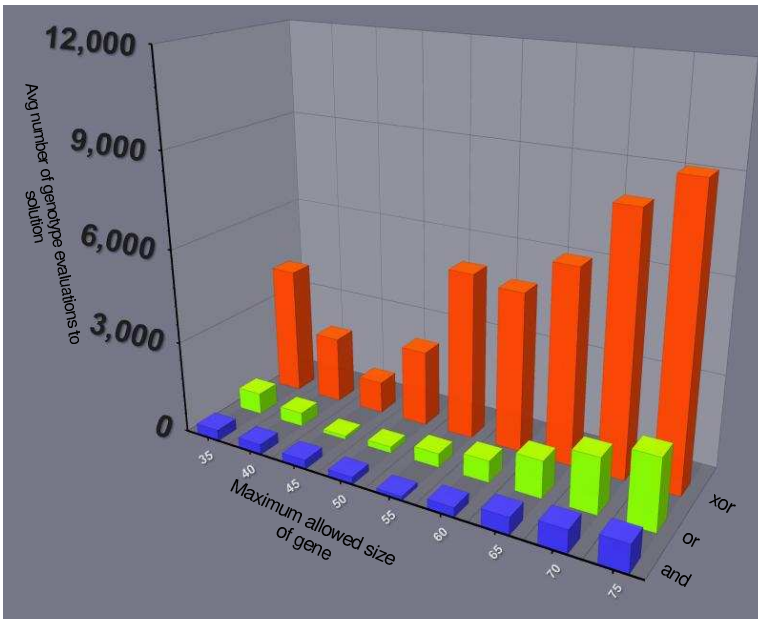

Figure 7.16: Relationship between average number of evaluation to either a maximum of 20 generations (equivalent to 20000 genotype evaluations) or to solution discovery and maximum size of genes in gene pool for each of the three boolean functions evolved. Each data point was obtained as the average of 50 runs. The settings for the system that do not vary are: $maxComp$= 1500 units, $ecosystemDefaultSize$= 1000

7.6.1.1 Allele Size Distribution

In an effort to obtain a sense of the distribution of allele sizes in the gene pool, we set up a variant of our implementation specifically for this purpose (we named it system 1). System 1 only evolves solutions to the xor problem. We provide the code and win32 executable for experimental system 1 online so that our results may be reproduced. It is available at the following address:

`http://code.google.com/p/gene-centered-ecosystem-framework-gp/source/browse/trunk#trunk/doc/franckbinardPhDthesis/experimental/system_1`. Figure 7.17 displays the size distribution of the alleles after one generation of a system 1 run when $maxGeneComp$ is set at 80 units of complexity. At the measurement point, there are 143 alleles in the gene pool and the mean allele fitness is 1.32 (out of a maximum of 4). Most alleles measure between between 60 and 80 units of complexity. We then repeated the experiment with $maxGeneComp$ set at 300 units of complexity. After 1 generation, there are only 85 alleles in the gene pool and the mean allele fitness is now only 0.73 (a substantial decrease in allele quality). Moreover, big alleles didn't survive the filter process as illustrated by the allele size distribution diagram (7.18). In fact, this version of the gene pool actually contains smaller alleles that the previous version, indicating that large low-quality alleles consumed most of the gene pool's capacity before being destroyed because of their low performance.

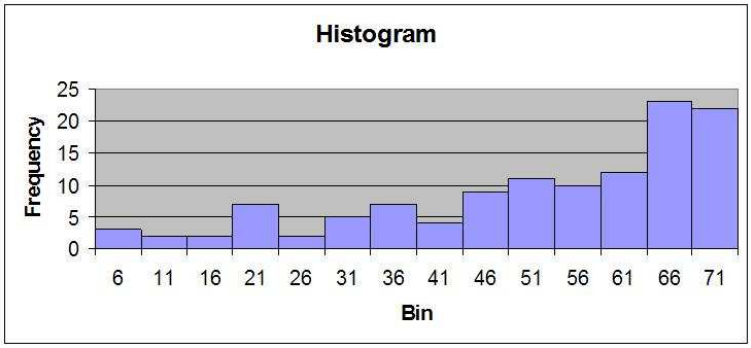

Figure 7.17: Distribution of allele size in gene pool after one generation when $maxGeneComp$ is set at 80 units of complexity. Results obtained using system 1. The settings for the system were: $maxComp$= 7000 units, $ecosystemDefaultSize$= 4000

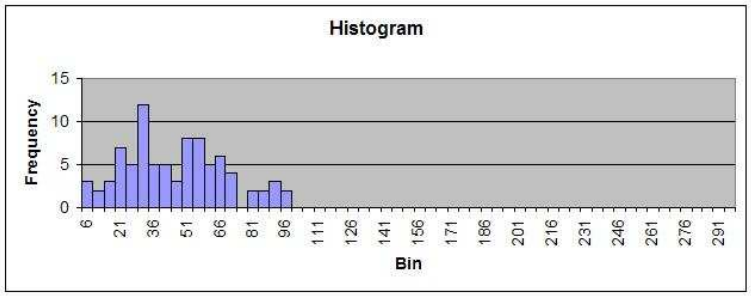

Figure 7.18: Distribution of allele size in gene pool after one generation when $maxGeneComp$ is set at 300 units of complexity. Results obtained using system 1. The settings for the system were: $maxComp$= 7000 units, $ecosystemDefaultSize$= 4000

7.6.2 Effect of Diversity Generating Measures on the Gene Pool

We've already mentioned that using the OCaml built-in $hash$ function, it is possible to compare the terms contained within genes by structure. The function takes a parameter $depth$ up to which it compares structures.

If two structures are similar (up to depth), the function returns the same value for both. The application of this method to differentiate genes results in the system's increased ability to make proper use of larger genes (around 60 units of complexity) to evolve solutions faster and more reliably. This demonstrates that diversity at the level of blocks has an important role to play in a modular GP system. Figure 7.19 presents the results of our experimentation with diversity at the gene level.

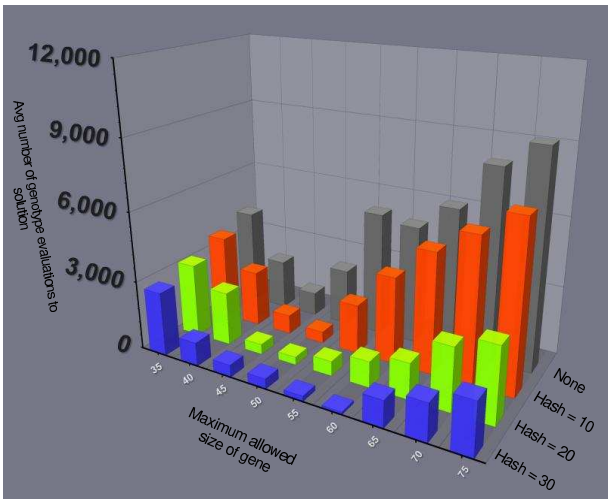

Figure 7.19: Relationship between average number of evaluation to either a maximum of 20 generations (equivalent to 20000 genotype evaluations) or to solution discovery, compared to the maximum size of genes in gene pool and to the hash differentiation factor for the evolution of the xor boolean function. Each data point was obtained as the average of 50 runs. The settings for the system that do not vary are: $maxComp$= 1500 units, $ecosystemDefaultSize$= 1000

7.6.3 Effect of Size of Genotype Population vs Size of Gene Pool

The number of genotypes in a given population is not as much of a factor as the allowed size of gene pool. Figure 7.20 shows the results obtained when comparing different size populations against different size gene pools when the maximum size of genes is kept fixed (55 units maximum gene size, hash diversity value of 10).

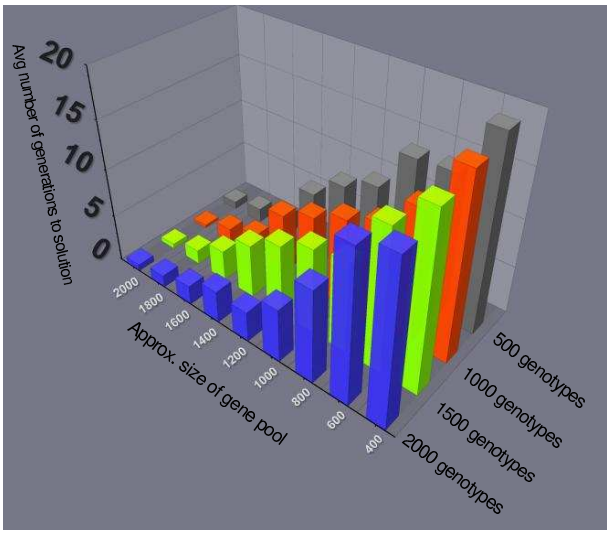

Figure 7.20: Relationship between average number of generations to convergence to solution and size of genotype population and maximum size of gene pool. Gene Pool Maximum Size of Individual Genes = 55 units, problem is xor, hash differentiator is set at 10

Chapter 8

Discussion

Some say that the essence of computing is in task automation, making computing a close cousin of calculating, a deterministic and thoughtless process. Computer scientists, however, are not designers of algorithms, but of languages. They build the tools that allow people to represent the world around them, not just to machines, but also to themselves and to other people. An abacus is a simple object. In fact, it is the simplicity of the abacus that is the source of its usefulness. As abacus users represent complex features of the real world as colored beads on strings, subsequent thoughtless, mechanical, speedy, error proof computations become possible. An abacus (the object) is really the wood and string representation of an abacus (the language), an alternative representation system for arithmetic expressions.

Databases, object oriented design, and functional programming are descendent of the abacus: linguistic tools that allow the representation of specific features of the real world so as to make calculation, communication and intellectual manipulation easy. A large part of this work is about the search for another linguistic tool: a language designed for the non-deterministic generation of computer programs. Just like the design of an object oriented language might go through an observation phase during which the general concept of objects in the world and how they interact with each other is studied, the design of Abstraction-Based Genetic Programming (ABGP) began with the same observation phase.

ABGP, the contribution of this work, is meant to facilitate the non-deterministic assembly of closed computational modules into complete computational structures. Its design required a mechanism to represent the modules (we found System F) and a mechanism to assemble them into more complex yet coherent structures (we used the Curry-Howard isomorphism). Biological life, as it turns out, is a system that arranges modules (genes) into complicated coherent structures (genomes). We therefore studied and reused the patterns of biological life. This is not to say that we blindly copied the patterns that we noticed- that would be impossible. There were many patterns that we didn't (and still don't) know, many that were simplified and many that were ignored. There were even some that we were able to improve on by exploiting the difference between physical reality, in which distance and energy place restrictions and the virtual computer reality medium in which many things can be abstracted. A computer system is able to represent a population of 5000 slightly identical objects in a manner more efficient than by storing 5000 separate object descriptions when the objects are 99.9% similar. In the real world however, there must exist 5000 different instances of pandas in order to

store 5000 nearly identical panda genomes. The next section describes the features of biological life that we attempted to reuse in the design of the ABGP system. The following section describes why System F seemed to be the most appropriate foundation for ABGP. Throughout this chapter, we highlight what we have done, and the directions that we feel future work should take.

8.1 THE LANGUAGE OF LIFE

> There is nothing mysterious about death, it is simply a way of keeping each species genetically flexible. Each of us is a temporary container for our immortal genes. We come to an end, but they go marching on - through our children - and, in the process, each generation sees a mixing of the genes that keeps offering new possibilities and enables our species to adapt to changing conditions. Sadly, this system works only if we as individuals are discarded after we have bred and reared our offspring. Or, as the saying goes:"Nature with its frugal eye asks only that we mate and die."

Desmond Morris, April 2008

8.1.1 Genes and Alleles

In general (with many exceptions), a gene is a container for a sequence of instructions. These instructions provide for the synthesis of a single protein product. Genes can abstractly be seen as beads aligned in sequence on strings. Each string of genes is called a chromosome. There may be several versions of the same gene. These versions are called alleles. In ABGP, we've reused this concept, with alleles designating closed System F computational blocks, encoding a specific computation.

8.1.1.1 Homeotic Genes

Some genes are more than sequences of instructions describing the synthesis of a single protein. There are more general purpose genes. Homeotic genes ([9], pg 251) are master-switch genes that guide the development of an organism by "turning on" a series of other, crucial genes. Homeotic genes are responsible for general phenotypic features, such as body plan partition and symmetry. Mutations to these types of genes often result in structural deformities such as one-eyed organisms or legs where antennae should be. In the computer analogy of the language of life, homeotic genes might be looping, branching or recursive features. Homeotic genes are still genes, and as such, are closed bead-like strings of dna. However, they still manage to have an effect on other closed dna units. Recently, it was discovered that similar homoetic genes control development in organisms as different as flies, mice, and humans, indicating that these genes are ancient, having first evolved hundreds of millions of years ago, and since then having been used in similar ways by a wide variety of organisms. The research presented in this thesis is founded on the belief that an homeotic gene system is crucial to GP and that it is the most promising path to the fabrication of GP systems endowed with the ability to construct programs capable of having the same levels of complexity that human made programs are capable of. Programs that, for example, include recursive, looping and complex data sub-structures. Figure 8.1 illustrates how ABGP is analogous to the homoeotic system. In the figure, an ABGP genotype contains the switch gene expressing $(gt\ time\ 10)$. It has type $[\Pi X.X \rightarrow X \rightarrow X]$ and corresponds to a

branching based on the comparison of the $time$ variable against the value 10. Much like the behavior of an homeotic gene, the gene will switch region 1 or region 2 in the genotype depending on the outcome of the comparison. As another example, we have already described the recursive application of an operation on the elements of a list in previous chapters. In this case, the list itself would be the homeotic gene. For example, an homeotic gene for a mammal might be the list [right leg location, left leg location, right arm location, left arm location] plugging into the operation "make limb bone structure". The mutation causing the homeotic gene to become: [right antenna location, left antenna location, right arm location, left arm location] would then replicate the effect observed in nature. There is an aspect of genetic descendence outlined by homeotic

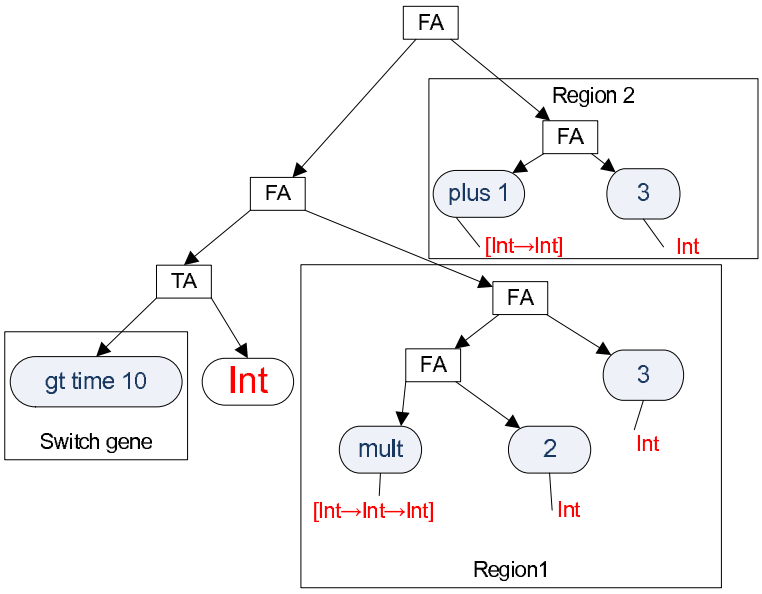

Figure 8.1: The ABGP analogy of an homeotic gene

genes that we would like to see future work explore. In nature, homeotic genes are extremely successful. They are so beneficial to their host organisms that they are transmitted down generations of organisms within species and are passed on to new descendent species as they get created. In a way, they point towards a two speed selection system: one at the level of genotypes and another at the level of genes.

Future Work: In biology, the smallest entity within the hierarchy of biological organization that is subject to natural selection is called the *unit of selection*. [21] proposed the gene as the biological unit of selection based on the observation that genes that improve the survival or reproductive chances of the organisms that carry them also improve their own chances of being passed on, so most of the time "successful" genes will also be beneficial to the organism. Adversely, carriers that are built from weaker combinations of genes tend to go extinct, reducing the set of genotypes in which they might be included. A gene that would cause an entity that carries it to die before it reproduces would be rapidly removed from the global gene pool because all potential carriers would die before they could transmit it to their offspring. Conversely, a gene that protects an organism against diseases improves the organism's survival. Dawkins's proposal leads to a *gene centric* selection mechanism. In the work presented in this thesis, we assign selection probabilities to genes at different times in their lives, but we never fully developed an explicit evolutionary system at the

level of genes. There is an implicit system as alleles that are not carried by any genotypes are removed from the gene pool, with the expectation that unsuccessful genes find themselves in unsuccessful genotypes and vice versa, but no effort to optimize the evolutionary process by adding a second selection layer at the gene level. We would like to see future work concentrate more on the idea of a fitness measure for genes, with an associated selection mechanism. Such a fitness measure might be calculated based on some average derived from the fitness of all the gene carriers. It might also be calculated based on the fitness of the best or worst genotypes that carry the gene. The number of genotypes that carry the genes, or the gene's age (in terms of generations) might also play a role in the determination of the quality of the gene as a computational component.

8.1.1.2 Inheritance of Combinations of Genes

Mendel's results on inheritance led him to propose the law of independent assortment, which stated that the alleles of one gene separate into gametes independently of the alleles for other genes ([9], pg 210). This is also the model that the reproduction process in ABGP follows. Early twentieth century biology, however, has discovered that genes were often inherited together, contradicting the law of independent assortment. In nature, gene linkage is a consequence of the grouping of genes into chromosomes in the genome. For example, humans have an estimated 100,000 genes, located on 23 pairs of chromosome, giving an average of 4348 genes per chromosome. The genes on a given chromosome tend to be linked and therefore inherited together. This is only a tendency because there is an intra-gene reshuffling at the chromosome level (called crossing-over, which has a different meaning than the term crossover commonly used in GP), but it is a very strong one. For example, the gene for body color is linked to the gene for wing length in fruit flies because both genes are located on the same chromosome. As a result of this tendency, many heritable genetic traits are the result of the expression of several genes working together to produce a single phenotypic trait.

Future Work: We think it may be beneficial to explore gene linkage within species in ABGP. The addition of such a scheme would require the addition of a partitioning scheme on the proof-species of the system. Each partition would be the equivalent of a chromosome and offspring would tend to be a blend of the partitions of their parents, rather than a blend of the individual alleles of their parents as the system is structured right now. The idea would be to encourage the formation of genetically stable combinations of genes within organisms, so that more complex sub-components of programs could evolve beyond what is possible by staying at the individual gene level.

8.1.1.3 Gene Flow

In GP, convergence can be seen as the equivalent of the biological phenomenon of *genetic bottleneck*, a term that describes low average genetic variations amongst the members of a population. Gene flow, which refers to the phenomenon of gene exchange between interbreeding populations is a cause of genetic bottleneck. Gene flow tends to make the gene pools of interbreeding populations more similar, acting as a convergent counterweight to the divergent effects of mutation, nonrandom mating, genetic drift and natural selection.

Future Work: Gene flow was considered in the design of ABGP. The main counter-measure to gene flow that was included in the design of ABGP is, of course, the speciation structure, but there were others. For

example, in section 6.6.2 we've described how genotypes of a population are initially created in such a way as to maximize their genetic differences. We are convinced that gene flow is a key issue in GP in general and in ABGP in particular. We would like to focus future work on this issue. There are two propositions that we would like to study further:

1. A formal measure of genotypic distance could be defined, where the distance between genotypes a and b (from the same species) would correspond to the number of alleles that are dissimilar in each. Given a population p, a new genotype would need to have its average distance to the other members of p greater than a certain threshold in order to be included in it.

2. It would be simple to include a dominant/recessive gene mechanism into the system. This could be done by changing the definition for genotypes provided in 6.6 to:

```
<genotype> ::=
             | "AlleleId" <int><int>
             | "TA" <genotype> <ty>
             | "FA" <genotype> <genotype>
```

   Each AlleleId branch would then link to two alleles, one dominant and one recessive, and the definition for alleles would be modified to include a value for dominance that could be used to decide which of two alleles is dominant and which is recessive. The effect of this would be to keep some low performing alleles in the gene pool longer as they might survive some extra generations by being on the recessive side of a genotype. This would in turn give them a greater chance of being chosen for mutation, potentially resulting in greater genetic diversity.

8.1.2 Species

Like genes, species are an integral part of the language of life. Species in biology are defined as groups of interbreeding natural populations that are isolated from other such groups. The phrase "interbreeding natural populations" includes populations that could interbreed if they were in contact with one another but do not because they have no opportunity to do so. There are limitations to this definition (in nature, some species can interbreed, producing hybrids, while others reproduce by asexual means) but a definition based on reproductive isolation works well in general. Biodiversity results directly from the splitting of one species into two or more species. Speciation acts as a counterweight to gene flow, which tends to keep populations similar to one another. In this work, we've used species as blueprints to organize sequences of genes into resembling coherent structures. As in nature, the same genes may be carried by organisms across species. Just like the mouse species may be considered a way to arrange combinations of genes so as to obtain a mouse organism, in ABGP, a species is a way to arrange System F blocks into similarly structured computer programs. To do this, we used the curry-howard isomorphism, which relates proofs to sets of programs.

8.1.3 Genotypes and Organisms

The genotype is the genetic constitution of a cell, an organism, or an individual (i.e. the specific allele makeup of the individual). In nature, organisms are the vessels that carry genes and are representative of the species to

which they belong. As such, within a species, all organisms share some characteristics, in particular at macro levels such as structure and general development, but differ on other characteristics. These differences are the reasons some organisms are better adapted to their environment within the same species. Adaptation, of course, is really just a name for the probability of the organism to pass on its gene onto the next generations.

1. By surviving to reproduction age, organisms demonstrate that the combination of genes they represent is viable in the environment in which they survive.

2. As sexual beings, one of the purpose of organisms the selection of a mate. is o selecxt a sexual mate.

In this work, organisms are the main statistical tools used to grade the genes they carry and the species they belong to.

Future Work: In nature, the size (measured either by volume or weight) of an organism is unrelated to the complexity of its genome. For example, while a cat is much smaller than an elephant, its genotype is in the same order of complexity in terms of the number of genes it contains. Neither the size of an organism nor the complexity of its genome are related to its evolutionary success, which is why life on earth spans such great diversity in terms of both. However, the physical world does establish some size related guidelines as to how a population of organisms belonging to the same species will evolve ([9]). For example, as the volume of an organism increases, its surface area (the area of the organism exposed to the outside) to volume ratio decreases. This makes it comparatively harder for large organisms to supply nutrients to their bodies as they contain more cells not exposed to the outside elements. For example, to supply its leaves with water and nutrients, a 20-meter tall tree must generate a pressure difference of about 3500 mm Hg between its roots and its upper leaves. The resulting observable trend is that larger organisms require comparatively more food intake than smaller organisms because more energy needs to be devoted to the movement of the nutrients through their bodies. This in turns plays with geographical considerations and population dynamics: larger organisms need more relative space to obtain more food, imposing predictable physical limits on their population size within a given land area. In our research, we haven't taken into account the sizes and structures of the phenotypes produced when assigning fitness values to their associated genotypes. Moreover, the ABGPS doesn't impose any upper bound on the size the population associated with a given species. One of the directions we would like to see further research explore is population dynamics. One idea would be to define a size metric on phenotypes (perhaps similar to the complexity metric defined for genes) and to use the average size of the phenotypes of a given population to impose restrictions on the number of individuals that it would be able to support at any given time. Bigger organisms would form smaller populations, live longer lives and produce less offspring, following the general guidelines observed in nature by biologists. This would potentially lead to the two evolutionary tracks (small, short lived, high reproductive rate, genetically very flexible vs. large, long lived, lower reproductive rate, less genetically variable) system that seems to exist in nature. This might open another path towards the goal of increasing overall genetic diversity in a given ecosystem.

8.2 SYSTEM F AS A GP LANGUAGE:CONCLUSION

Almost all variations of the GP paradigm, including the one represented in this thesis, represent the individuals of the system as trees: nodes containing information related to an operation and linked to other nodes

containing the inputs needed by the operation carried by the parent node. The trees are assembled together from units defined by the system designer and specific to the problem. A GP system is not a random search algorithm, however, many of its mechanisms use randomization. This includes both the procedures that produce new trees from nothing and the procedures that build new trees from old ones. The following issues arise almost immediately:

Question I: In a system that uses a randomized procedure to produce trees that will eventually be interpreted as programs, how can we reliably insure that the trees that are built always make sense from a computational point of view?

Question II: In a system that includes a finite amount of primitive operations and a relatively simple scheme for combining them, how can we insure that it is possible to assemble the computational tools needed to represent solutions to a specific problem.

8.2.1 Producing Genotypes that "Make Sense"

We've already described how the addition of a typing system resolves the "making sense" issue. There are many arguments in the literature for using type systems in GP. There is one however that is seldom mentioned: Types may also be used as partial specifications as to the kind of program the GP system should evolve. "Making sense" in this context means evolving something that can be reliably placed in a wider context. How can operations on lists be evolved ? How do we specify that we wish to evolve a function that takes a tree structured information set containing humidity values and a function expressing some empirically obtained relationship on humidity and temperature as input and outputs a temperature? How do we change the type (that's what it is) of what we want to evolve to something else without rebuilding the whole system? Types let us specify the expected result of the GP system. Types also provide the ability to classify computational objects, thus providing specifications on how different objects may be combined in a program. The automatic classification system that types provide is a valuable tool in a GP context where a system is expected to combine different computational objects in a random manner. In human programming, imposing a type structure on a programming language makes many errors that would otherwise be found by hand searching detectable by a compiler as type errors. Similarly, a type structure lets the GP system differentiate programs that make sense from the ones that don't. The system's primitives always have a direct meaning in the domain in which the problem that is being solved by the system lives. That meaning is typed because the universe is typed. For example, a GP system might include a moving average primitive function that is meaningful only when applied to a specific kind of time series intervals. A type system allows the designer to define a type for the time series interval, making the moving average operation usable only with that type.

8.2.2 Expressiveness

The remaining issue is one of *expressiveness*. Pinning down the meaning of expressiveness in the context of a programming language is not an easy task. The notions of "expressive power" or "expressiveness" of knowledge representation languages can be found in most papers on knowledge representation, but these terms are usually just employed in an intuitive sense.

If expressiveness is merely defined by the ability of the language to express as many things as possible and if there is a link between the expressiveness of the representation scheme of a GP system and its performance, then simply adding syntax to the system's vocabulary would make GP systems perform better. But we already know that is not the case. In fact the more syntax that is added to the GP system's vocabulary, the larger its search space becomes and the harder it becomes for it to find a solution. So what is needed is a language that contains very few symbols, from which the largest proportion as possible have a direct meaning in the problem's domain. But that language should still be able to express as many different complicated patterns and structures as possible. In many problems, GP has produced solutions better than any produced by rational design. In other cases, it has failed. What turns out to be crucial to the success of a GP system is how the genotypes are represented as data structures. This is known as the *representation problem*. There are several properties of System F which motivated our decision to use it as the basis for a representation scheme for GP:

- **Typing structure:** System F is a polymorphically-typed λ-calculus. It has variables ranging over functions and data, but it also has variables ranging over types. This allows the definition of true parametric polymorphic functions, making the language very expressive while still maintaining full static type-safety.

- **Language lightness:** The pure System F syntax uses a very small set of symbols. Yet, it is possible to represent and use many useful computer science structures (recursion, lists, trees, booleans, looping) from within the language without adding anything to it. In fact, many functions and terminals that need to be defined for GP problems in other systems can be eliminated for the same problem when the GP uses System F as its underlying representation scheme.

- **Tree-like structure:** System F programs are similar in structure to programs written in other functional languages such as Lisp. This tree-like structure is generally considered a natural choice for GP.

- **Expressive power:** System F's abstraction power makes the system very expressive. Under the Curry-Howard isomorphism [40], System F corresponds to a second-order logic and can therefore describe all functions which are provably total in second-order logic.

- **Second-order logic isomorphism:** The isomorphism provides a dual meaning to the type structure. For example, the type

  $$[\Pi X.X \to X \to X]$$

  can also be read as the second-order proposition:

  $$\forall X.X \to X \to X$$

  Adding the gene (or type) $[A]$ to an ABGP gene pool corresponds to adding the proposition A as an axiom. The second-order derivation rules can be used directly to produce new types. This becomes very useful when used in conjunction with the next item.

- **Strong Normalization:** System F programs always terminate, and they always evaluate to the same thing no matter the evaluation scheme used.

Future Work, Correctness Proofs: We would like to see some futurr work explore the possibility of evolving correctness proofs in conjunction with programs. This could be done by introducing the use of a more complex typing system, in which types contains additional information, or by associating each allele in the gene pool with a pre and post condition.

8.3 FINAL THOUGHTS

It is clear that the power of System F comes from both its simplicity and expressiveness. It is a language that doesn't use many symbols, doesn't have many rules and yet is naturally capable of expressing many computations in many different styles. It does recursion and even allows us to define and work with any of the structures usually used by programmers. All this from within the system and using only two abstraction operations. Even greater simplicity and expressiveness can be found in the un-typed version of the calculus. Unfortunately the un-typed version of the λ-calculus is less suitable in a context in which programs are created randomly because it lacks the normalization property.

There is no representation that can serve all purposes or make every problem equally approachable, but problem solving can often be simplified by an appropriate choice of knowledge representation. In this document it is argued that System F might be an appropriate representation choice for some problems solvable by genetic programming.

In the last decades, many new exciting areas of computer science have appeared. From research into emergence to cellular automaton by way of chaos theory, the one paradoxical observation that seems to have started many of these new research pathways is that very simple systems with very simple rules can produce very complicated behavior. Of course, most simple systems produce simple behavior, but every once in a while, a game of life [27] or a recursive function such as:

$$Q(1) = Q(2) = 1\ ,\ Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2))$$

taken from page 137 of [36] proves itself capable of producing very complicated behavior. Often, the people who are responsible for the publication of these systems will say that they "discovered" the system rather than they "invented" it. System F's simplicity compared to the wide range of things that it can be used to represent in it seems to belong to that category.

Acknowledgments: This work is made possible by my supervisor, Amy Felty, who provided financial support but, also (and more importantly) careful and patient reading, re-reading, comments and advice.

Bibliography

[1] S.F. Allen, R.L. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The Nuprl Open Logical Environment. *Automated Deduction-Cade-17: 17th International Conference on Automated Deduction, Pittsburgh, Pa, Usa, June 2000: Proceedings*, 2000.

[2] L. Altenberg. *The Evolution of Evolvability in Genetic Programming*. Institute of Statistics and Decision Sciences, Duke University, 1993.

[3] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

[4] HP Barendregt. The Lambda Calculus, its Syntax and Semantics. 1981.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*. Springer, 2004.

[6] F. Binard and A. Felty. An abstraction-based genetic programming system. *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2415–2422, 2007.

[7] Scott Brave. Using genetic programming to evolve recursive programs for tree search. In *Proceedings of the Fourth Golden West Conference on intelligent Systems*, pages 60–65, NC, 1995. International Society for Computers and Their Applications, Raleigh.

[8] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *The Journal of Symbolic Logic*, 40(3):470, Sep 1975.

[9] M. Cain. *Discover Biology*. WW Norton & Co Ltd, 2000.

[10] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[11] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[12] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[13] A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[14] Chris Clack and Tina Yu. Performance enhanced genetic programming. In Peter J. Angeline, Robert G. Reynolds, John R. McDonnell, and Russ Eberhart, editors, *Evolutionary Programming VI*, pages 87–100, Berlin, 1997. Springer.

[15] T. Coquand and G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics*. Institut National de Recherche en, Informatique et en Automatique, 1985.

[16] Nichael Lynn Cramer. A representation for the adapative generation of simple sequential programs. In *International Conference on Genetic Algorithms and their Applications [ICGA85]*, Pittsburgh, 1985. CMU.

[17] HB Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[18] H.B. Curry, R. Feys, W. Craig, and W. Craig. *Combinatory logic, vol. 1*. North-Holland, 1958.

[19] Richard Dallaway. Genetic programming and cognitive models. Technical Report CSRP 300, School of Cognitive & Computing Sciences, University of Sussex,, Brighton, UK, 1993. In: Brook & Arvanitis, eds., 1993 The Sixth White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex.

[20] L. Davis. *Handbook of genetic algorithms*. Van Nostrand Reinhold.

[21] R. Dawkins. *The Selfish Gene: New Edition*. Oxford University Press, Oxford, England, 1989.

[22] NG de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125, pages 29–61, 1970.

[23] P. de Groote. *The Curry-Howard isomorphism*. Academia-Erasme.

[24] R. Feys. Les logiques nouvelles des modalites. *Revue Neoscholastique de Philosophie*, 40:517–553, 1937.

[25] David B. Fogel. *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1998.

[26] Richard Forsyth. BEAGLE A Darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.

[27] M. Gardner. Mathematical Games: The Fantastic Combinations of John Conways New Solitaire Game Life. *Scientific American*, 223(4):120–123, 1970.

[28] J.Y. Girard. Une extension de l'interprétation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstadt, editor, *Proc. 2nd Scandinavian Logic Symp.*, pages 63–92, Amsterdam, 1971. North-Holland.

[29] M. R. Glickman and K. Sycara. Evolvability and static vs. dynamic fitness. In Maley, editor, *Workshop Proc. of Artificial Life VII*, August 2000.

[30] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[31] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[32] J.J. Grefenstette and J.E. Baker. How genetic algorithms work: a critical look at implicit parallelism. *Proceedings of the third international conference on Genetic algorithms table of contents*, pages 20–27, 1989.

[33] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 271–278, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[34] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In P. J. Angeline and Jr K. E. Kinnear, editors, *Advances in Genetic Programming II*, pages 359–376. MIT Press, Cambridge, MA, 1996.

[35] J.R. Hindley and J.P. Seldin. *Introduction to combinators and &lgr;-calculus*. 1986.

[36] D.R. Hofstadter, M.C. Escher, K. Gödel, and J.S. Bach. *Godel, Escher, Bach: An Eternal Golden Braid*. Vintage Books USA, 1989.

[37] J.H. Holland. Adaptation in Natural and Artificial Systems. 1975. *Ann Arbor MI: University of Michigan Press*.

[38] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[39] W.A. Howard. The formulae-as-types notion of construction. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, 1980.

[40] William A. Howard. The formulae-as-type notion of construction, 1969. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[41] Jianjun Hu. *Sustainable Evolutionary Algorithms and Scalable Evolutionary Synthesis of Dynamic Systems*. PhD thesis, Michigan State University, 2004.

[42] Lafont J.Y. Girard and Taylor. *Proofs and Types*. Cambridge University Press., 1997.

[43] K.E. Kinnear and P.J. Angeline. *Advances in Genetic Programming*. MIT Press, 1994.

[44] S.C. Kleene and J.B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[45] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the 11th International Conference on Artificial Intelligence.*, volume I, pages 768–774, Detroit, 1989. MI. Morgan Kaufmann.

[46] John R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI, Herndon, Virginia, USA*, pages 819–827. IEEE Computer Society Press, Los Alamitos, CA, USA, 6-9 1990.

[47] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[48] John R. Koza. The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In Branko Soucek and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–321. John Wiley, New York, 1992.

[49] John R. Koza. Simultaneous discovery of reusable detectors and subroutines using genetic programming. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 295–302, University of Illinois at Urbana-Champaign, 17-21 1993. Morgan Kaufmann.

[50] John R. Koza. Genetic programming version 2. Submitted for inclusion in the Encyclopaedia of Computer Science and Technology, 1997.

[51] J.R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 1:768–774, 1989.

[52] P. J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, 1965.

[53] W.B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.

[54] H. Lauchli. An abstract notion of realizability for which intuitionistic predicate calculus is complete. *Intuitionism and Proof Theory*, page 227, 1970.

[55] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Napoli, 1984.

[56] R. Milner. A proposal for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 184–197. ACM New York, NY, USA, 1984.

[57] J.C. Mitchell. *Foundations for programming languages*. MIT press Cambridge, Mass, 1996.

[58] T Mitchell. *Machine Learning*. McGraw-Hill, New York, 1996.

[59] D.J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[60] U. M. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. Technical Report 94-02-001, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1992.

[61] U.M. O'Reilly. *An analysis of genetic programming*. Carleton University Ottawa, Ont., Canada, Canada, 1995.

[62] F. Pfenning. Logic programming in the LF logical framework. *Logical Frameworks*, pages 149–181, 1991.

[63] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[64] R. Pollack. The Theory of LEGO: a proof checker for the Extended Calculus of Constructions. *Doctor of Philosophy thesis, University of Edinburgh*, 1994.

[65] M. Randall, C. Thorne, and C. Wild. A standard comparison of adaptive controllers to solve the cartpole problem. *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 61–65.

[66] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[67] J.P. Rosca. Analysis of complexity drift in genetic programming. *Genetic Programming*, pages 286–294, 1997.

[68] AL Samuel. Some studies in machine learning using the game of checkers, Reprinted in EA Feigenbaum & J. Feldman (Eds.)(1963). *Computers and thought*, 1959.

[69] Jurgen Schmidhuber. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 14 May 1987.

[70] Hans-Paul Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[71] S.F. Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980.

[72] L. Spector. Simultaneous evolution of programs and their control structures. In P. J. Angeline and Jr K. E. Kinnear, editors, *Advances in Genetic Programming II*, pages 137–154. MIT Press, Cambridge, MA, 1996.

[73] C. Strachey. Fundamental concepts in programming languages. Technical report, Lecture Notes from the International Summer School in Computer Programming, Copenhagen, Denmark, 1967.

[74] W.A. Tackett. Genetic Programming for Feature Discovery and Image Discrimination. *Proceedings of the 5th International Conference on Genetic Algorithms table of contents*, pages 303–311, 1993.

[75] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[76] P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 1995.

[77] PA Whigham. A Schema Theorem for context-free grammars. *Evolutionary Computation, 1995., IEEE International Conference on*, 1.

[78] P.A. Whigham. Grammatical bias for evolutionary learning. 1996.

[79] D. Whitley. The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best. *Proceedings of the third international conference on Genetic algorithms table of contents*, pages 116–121, 1989.

[80] T. Yu and P. Bentley. Methods to Evolve Legal Phenotypes. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 280–291, 1998.

[81] Tina Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. PhD thesis, University of London, 1999.

[82] Tina Yu. Polymorphism and genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038, pages 218–233, Lake Como, Italy, 18-20 2001. Springer-Verlag.

[83] Tina Yu. A higher-order function approach to evolve recursive programs. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 7, pages 93–108. Springer, Ann Arbor, 12-14 May 2005.

[84] Tina Yu, Shu-Heng Chen, and Tzu-Wen Kuo. Discovering financial technical trading rules using genetic programming with lambda abstraction. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 2, pages 11–30. Springer, Ann Arbor, 13-15 May 2004.

[85] Tina Yu and Chris Clack. Recursion, lambda abstractions and genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 422–431, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[86] B.T. Zhang and H. Muhlenbein. Balancing Accuracy and Parsimony in Genetic Programming. *Evolutionary Computation*, 3(1):17–38, 1995.

Index

Appendix A

The ABGP Software

In this appendix, we provide commented screenshots of the application that was built for the purpose of this research. The reader may also use these to “get a feel” for the process of evolving a program using ABGP.

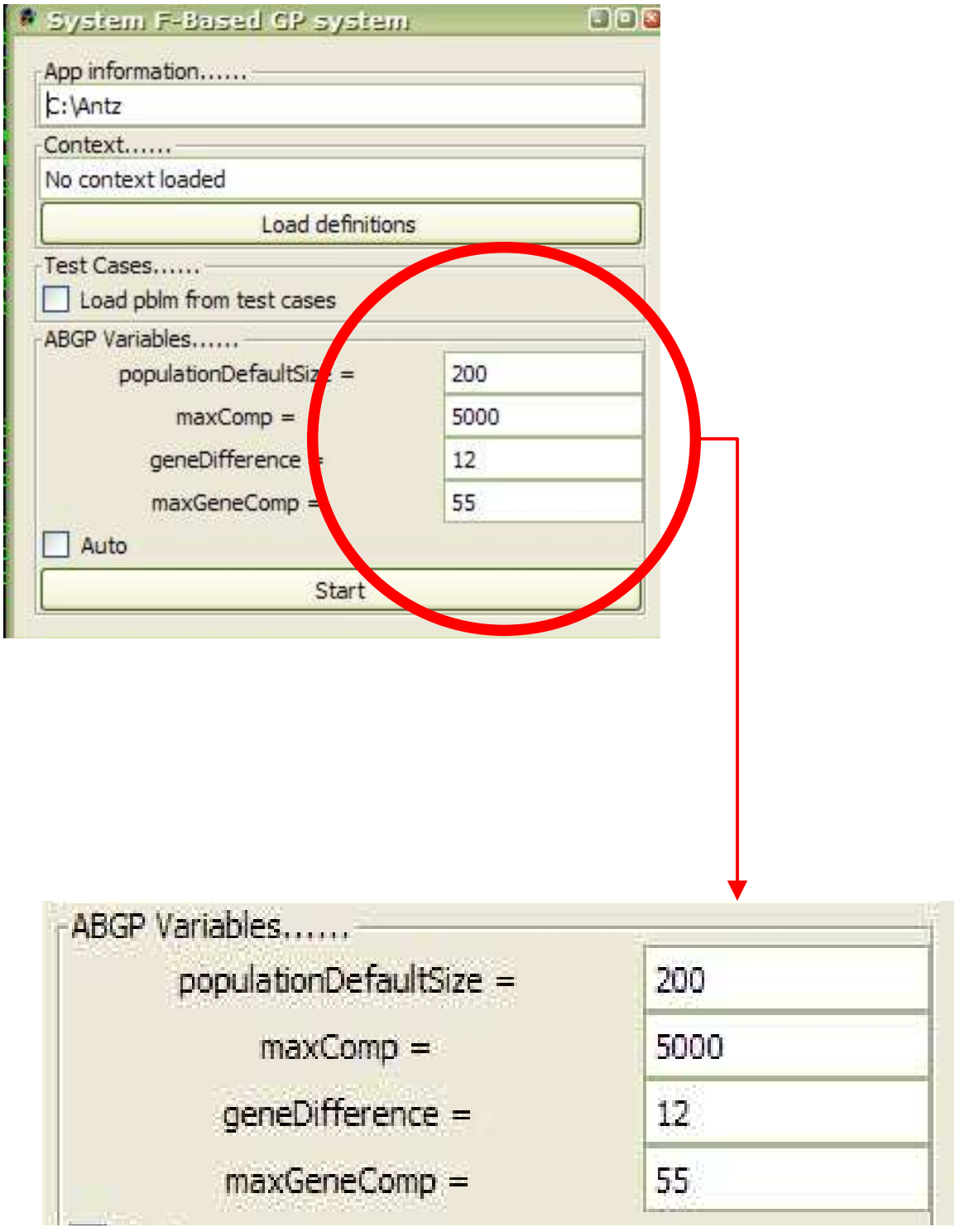

Figure A.1: The application's entry point and the user specified parameters for the GP run

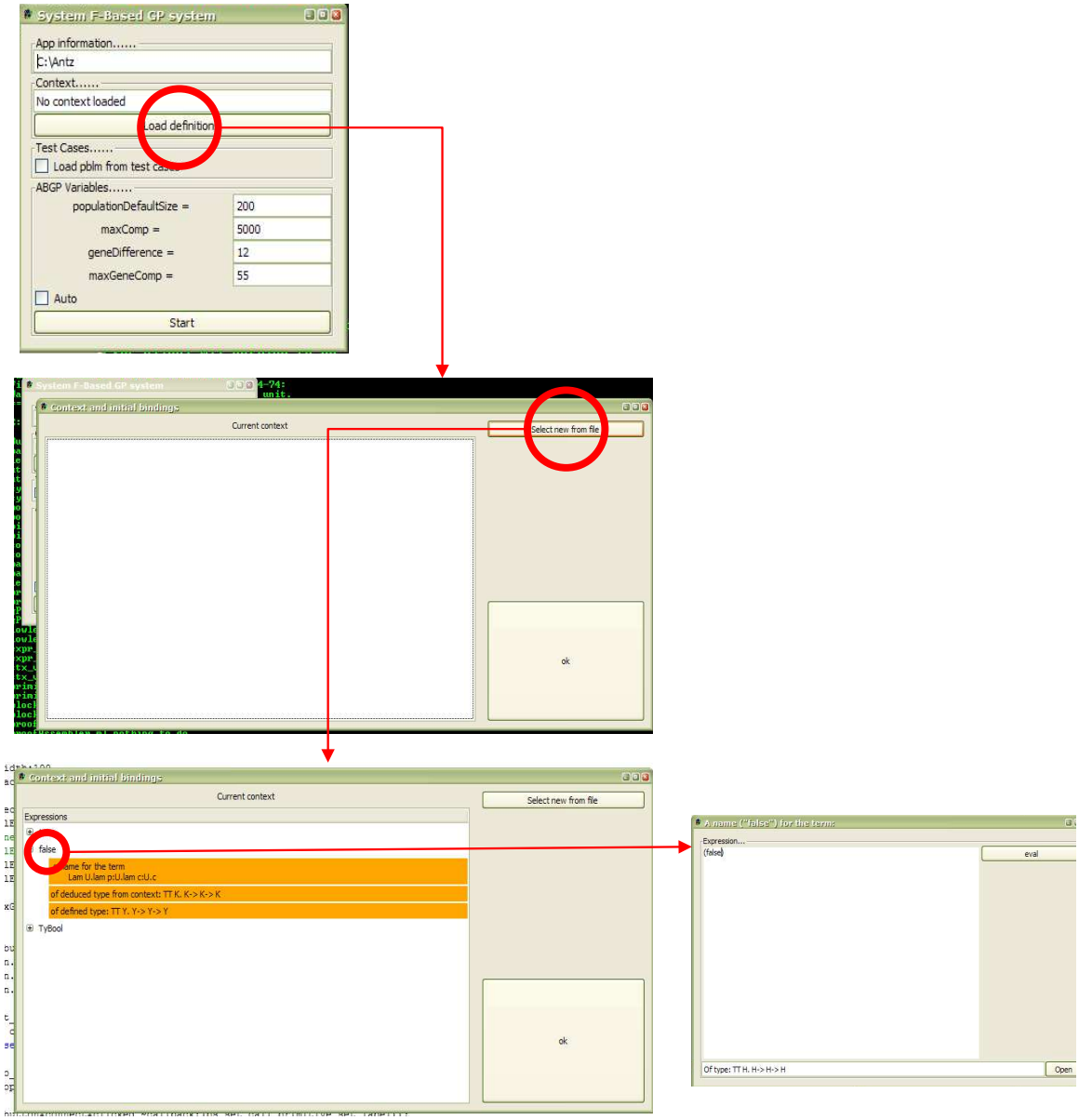

Figure A.2: The application's entry point, and loading a context

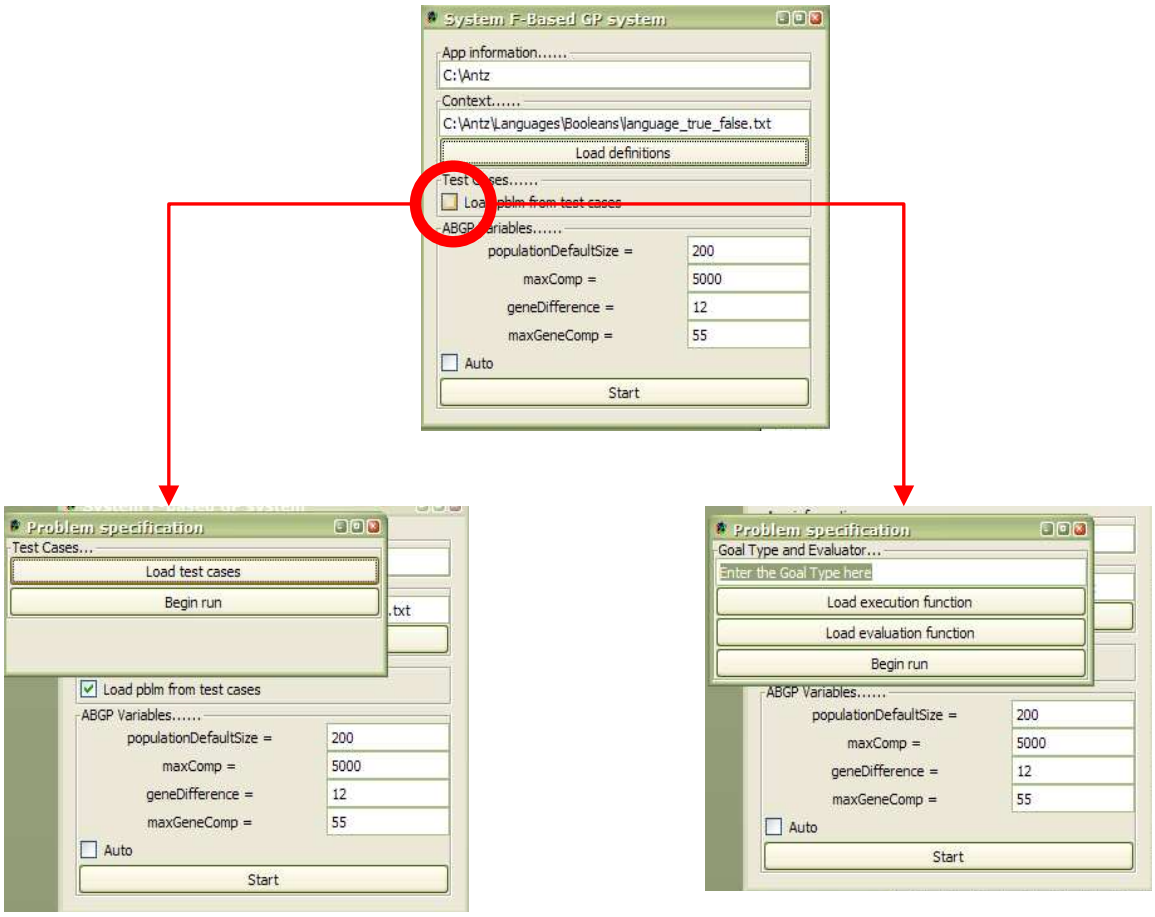

Figure A.3: A problem may be specified either by test cases or by a custom method, such as for the ant application

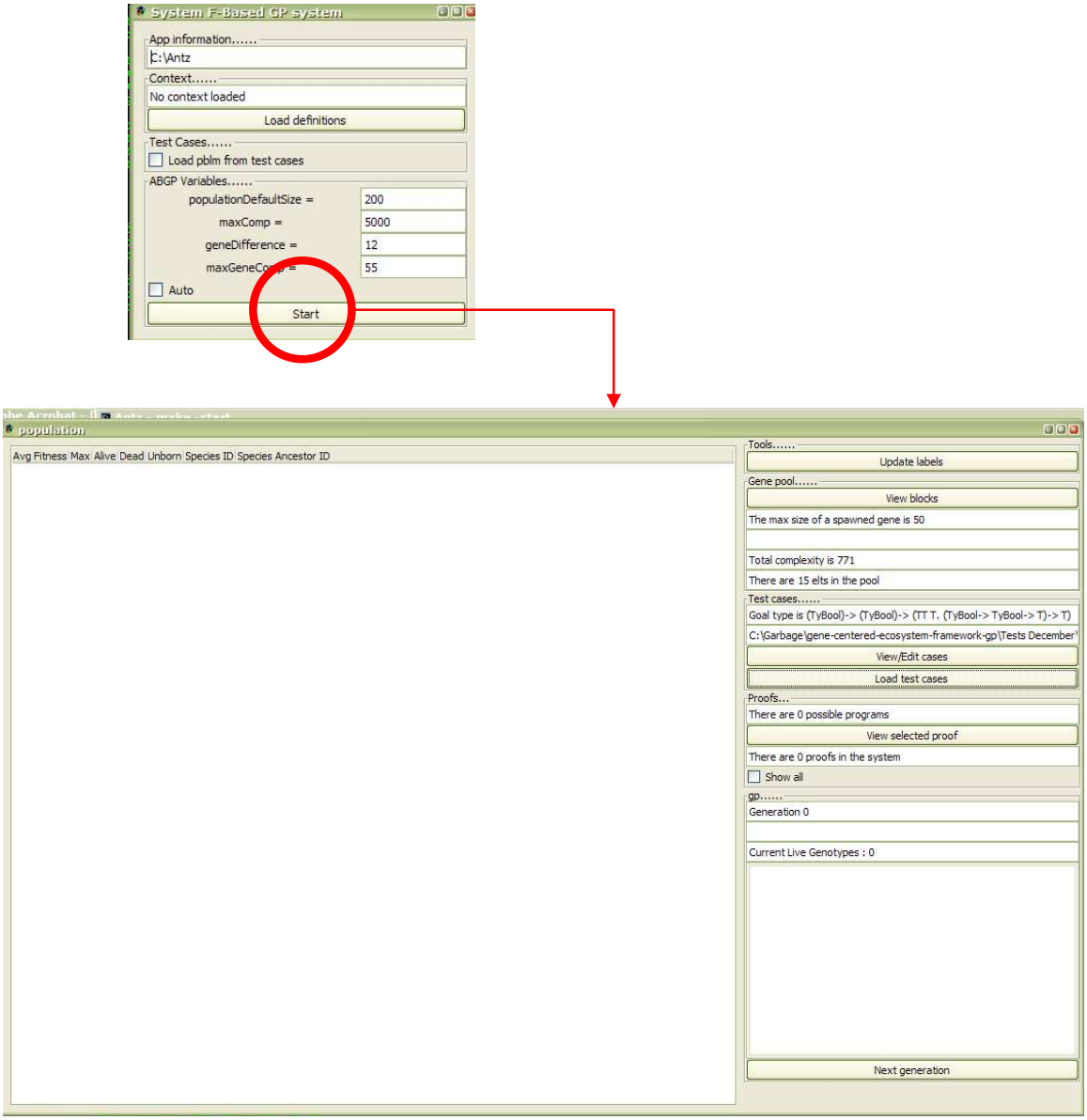

Figure A.4: Once all the necessary input values have been supplied, the user may access the main GP screen and start the run

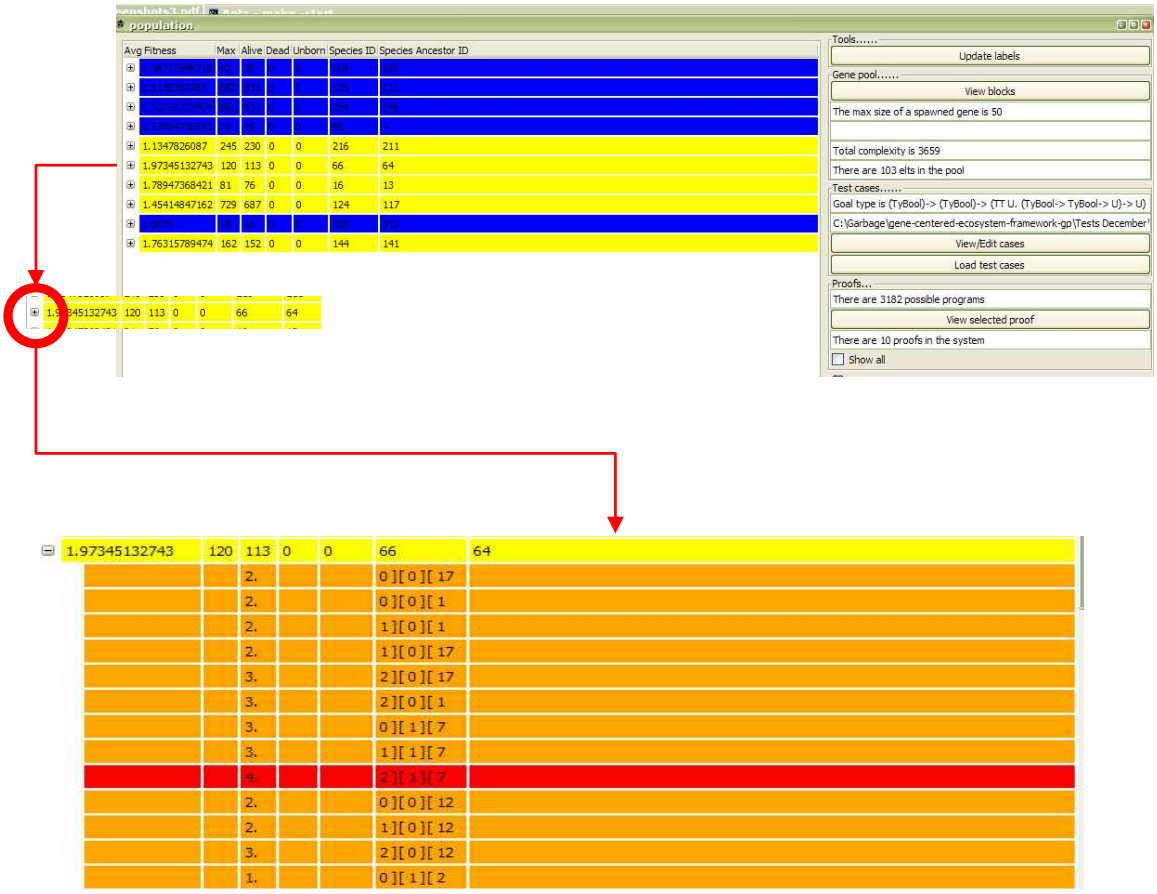

Figure A.5: In between each generation, the user can see the species and the genotypes. Each row on the main panel represents a species. By clicking on the plus to the left of the row, the user can see all the genotypes that are currently alive for that species. Yellow colored rows are species that contain a solution. Red colored rows are genotypes that are solutions.

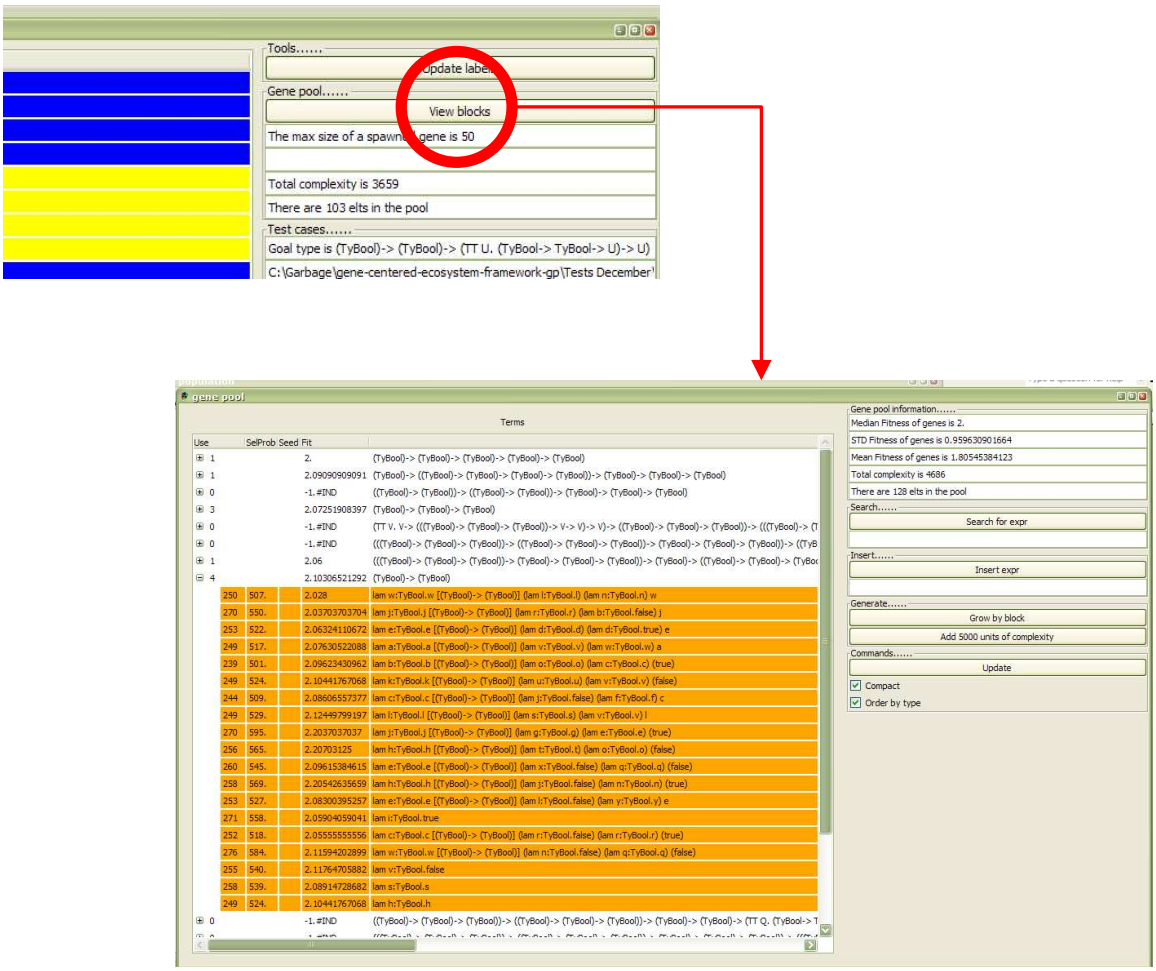

Figure A.6: In between each generation, the user may also browse through the gene pool by clicking on the "View blocks" button.

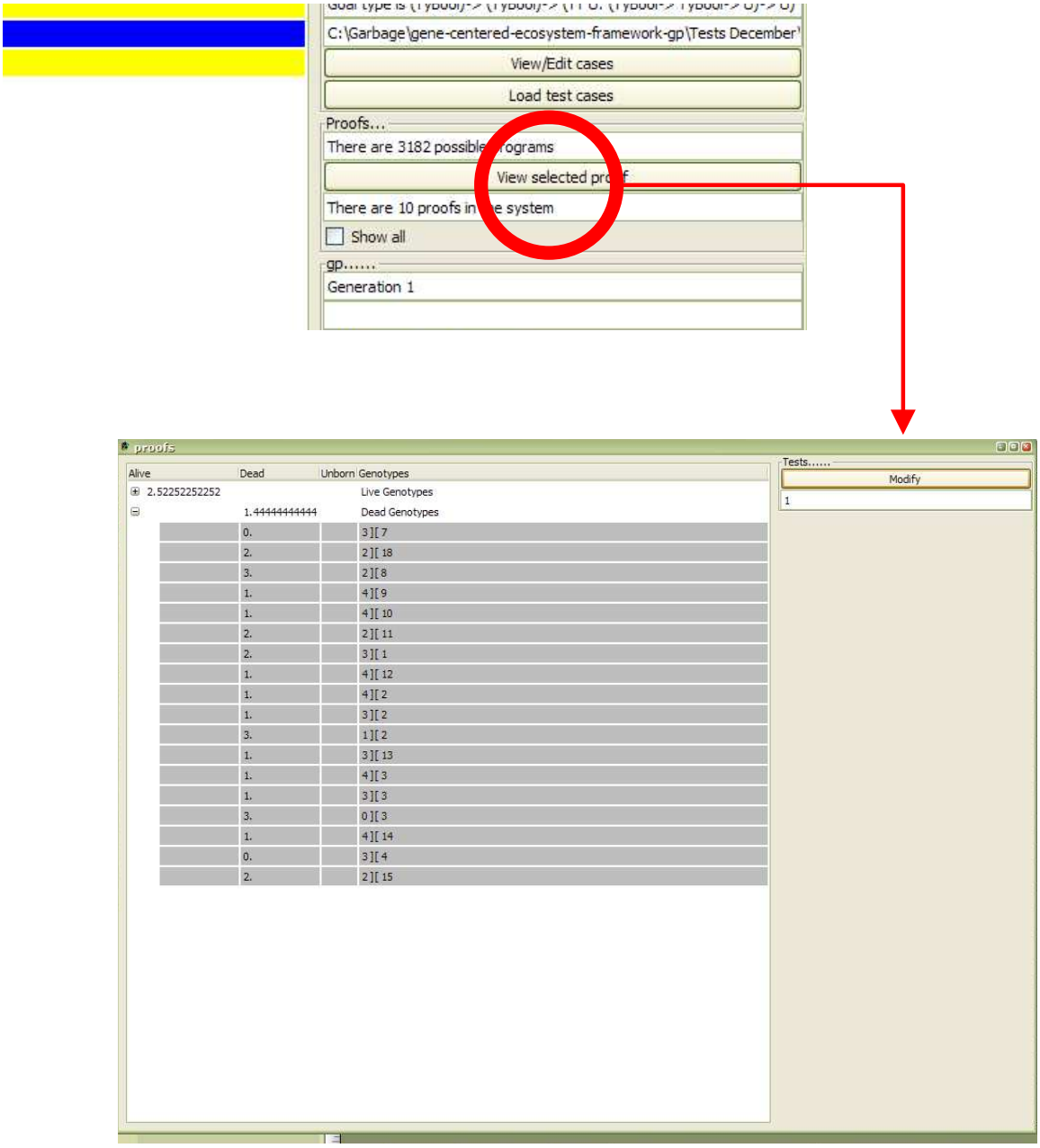

Figure A.7: Selecting a proof and clicking clicking on the "View selected proof button" brings user to a species specific view from which are displayed the genotypes both alive and dead of the selected species, while double-clicking on the row for the species creates a gif representation of the species, such as those used in the result chapter

Appendix B

The $ABGP$ important function

Program B.0.1 The *make_constructors* function

```
let make_constructors ctx ty =
    let make_abs (name,ty) =
        "lam " ^ name ^ " : " ^ (string_of_type ctx ty Alpha) ^ "." in

    match ty with
    |TyArr (ty1,ty2) when is_data_type ctx ty1 ->
        (* inner_term (below) corresponds to the term  *)
        (* (x [ty2]) in which x is a variable of type  *)
        (* ty1                                         *)
        let inner_term = TmTApp((TmVar 0), (typeShift 1 ty2)) in
        let inner_ctx, varName1 =  fresh_fvar ctx ty1 in
        let cons_types =
            get_constructor_arguments ctx ty2
            (typeShift (-1) (typeof inner_ctx inner_term)) in

        (* (typeof (x [ty2])) is a constructor for [ty2] *)
        (* each of the constructor arguments is          *)
        (* associated with a new function name           *)
        (* name variable                                 *)
        let names_types = (List.map (fun ty -> fresh_lcid( ),ty) cons_types) in

        (* which allows the string form construction of  *)
        (* the common inner-bodies for all the           *)
        (* constructors                                  *)
        let final_inner_term = ref (string_of_term inner_ctx inner_term Alpha) in
        List.iter
            (fun (name, _) -> final_inner_term := !final_inner_term ^ " " ^ name)
            names_types;
        final_inner_term := make_abs(varName1,ty1)^ !final_inner_term;

        (* All the permutations of constructors can now *)
        (* be constructed  in their string form         *)
        let constructors = ref [] in
        List.iter (fun name_type_lst ->
            constructors :=
                (String.concat ""
                    ((List.map make_abs name_type_lst) @ [!final_inner_term]))::
                !constructors)  (permutations names_types);

        (* and returned in their term representation    *)
        List.map (fun str -> ter(parse_string str ctx))  !constructors

    |_->failwith "non constructible type"
```

Appendix C

Mutation Rules for Genes

When a new set of genes is created by seed mutation, the genes from the descendant set that are not already in the gene pool and that do not violate size constraints (see Gene Complexity, section 6.4.1) are added to the gene pool. There are many possible mutation operations that may be applied to a seed to generate a set of descendant genes. Below is a description of the mutation operations we use to produce new genes: The set of rules that we use in this research was obtained by trial and error.

C.1 Generating a Descendent Set of Expressions from a Seed Term

C.1.1 Context

Given a context C ,below are listed the term descendence rules.

1. If the seed expression is the term t, of type $[\Pi X.V]$, then one can generate the descendant set composed of all expressions $(t\ A)$, where A is a type that exists in t's context.
2. If the seed expression is the term t, of type $[A \rightarrow B]$, then one can generate the descendant set of of all expression $(t\ a)$, where a is a term of type A that exists in t's context or as a gene expression in the gene pool.
3. If the seed expression is the term t, of type $[T]$, that contains a constant a^A, then one can generate the descendant set of one element $\{(\lambda x^A.t\{a := x\})$ of type $[A \rightarrow T]\}$
4. If the seed expression is the term t, of type $[T]$ that contains a type constant $[A]$, but does not contain any term constant of type $[A]$ then one can generate the descendant set of one element: $\{(\Lambda X.t\{A := X\})$ of type $[\Pi X.T\{A := X\}]\}$
5. If the seed is the term $\lambda x_1^{X_1} \lambda x_2^{X_2}.t$, then one can generate the descendent set of one element $\{\lambda x_2^{X_2} \lambda x_1^{X_1}.t\}$
6. If the seed is the term $\lambda x^V.u$ then one can generate the descendent set obtained by:

(i) applying any of the listed applicable descendence rules to obtain D_1, the descendent set of u with context $C \cup \{x^V\}$

(ii) for every expression t in D_1, adding $\lambda x^V.t$ as an element of the descendent set of $\lambda x^V.u$

7. If the seed is the term $\Lambda X.u$ then one can generate the descendent set obtained by:

(i) applying any of the listed applicable descendence rules to obtain D_1, the descendent set of u with context $C \cup \{X\}$.

(ii) for every expression t in D_1, adding $\Lambda X.t$ as an element of the descendent set of $\Lambda X.u$

Appendix D

Gene Pool Examples

D.1 Example Boolean Function

Context:

```
/*---------------------------------------*/
true = Lam X . lam x : X . lam y : X . x;
false = Lam X . lam x : X . lam y : X . y;
TyBool = TT X . X -> X ->X;
/*****************************************/
```

The following gene pool was created randomly using the context defined above. It contains 70 elements and has a total complexity of 2923 units. The generation procedure was instructed to refuse any gene with complexity greater than 85 units.

```
type: (TyBool)-> (TT E. ((TyBool)-> E)-> E)
   id:0  lam h:TyBool.Lam R.lam r:(TyBool)-> R.r h
type: (TyBool)-> ((TyBool)-> (TT W. (((TyBool)-> (TyBool))-> W)-> W))-> ((TyBool)-> (TT R. (((TyBool)-> (TyBool))-> R)-> R))-> (TyBool)-> (TT U. (((TyBool)-> (TyBool))->
   id:0  lam g:TyBool.g [(TyBool)-> (TT W. (((TyBool)-> (TyBool))-> W)-> W)]
type: (TyBool)-> ((TyBool)-> (TT M. ((TyBool)-> M)-> M))-> ((TyBool)-> (TT T. ((TyBool)-> T)-> T))-> (TyBool)-> (TT M. ((TyBool)-> M)-> M)
   id:0  lam q:TyBool.q [(TyBool)-> (TT R. ((TyBool)-> R)-> R)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> ((TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-
   id:0  lam g:TyBool.lam r:TyBool.r [(TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)]
type: (((TyBool)-> (TyBool))-> (TyBool)-> (TT F. (((TyBool)-> (TyBool))-> F)-> F))-> (TyBool)-> (TT W. (((TyBool)-> (TyBool))-> W)-> W)
   id:1  lam r:((TyBool)-> (TyBool))-> (TyBool)-> (TT F. (((TyBool)-> (TyBool))-> F)-> F).r (lam q:TyBool.false)
   id:0  lam p:((TyBool)-> (TyBool))-> (TyBool)-> (TT V. (((TyBool)-> (TyBool))-> V)-> V).p (lam r:TyBool.true)
type: ((TyBool)-> (TT A. (((TyBool)-> (TyBool))-> A)-> A))-> (TyBool)-> (TT Q. (((TyBool)-> (TyBool))-> Q)-> Q)
   id:0  lam n:(TyBool)-> (TT S. (((TyBool)-> (TyBool))-> S)-> S).n
type: (TyBool)-> ((TyBool)-> (TyBool)-> (TT I. (((TyBool)-> (TyBool))-> I)-> I))-> (TyBool)-> (TT O. (((TyBool)-> (TyBool))-> O)-> O)
   id:0  lam l:TyBool.lam y:(TyBool)-> (TyBool)-> (TT V. (((TyBool)-> (TyBool))-> V)-> V).y l
type: (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TT Y. (((TyBool)-> (TyBool))-> Y)-> Y))-> (TyBool)-> (TT I. (((TyBool)-> (TyBool))-> I)-> I)
   id:0  lam e:TyBool.lam l:((TyBool)-> (TyBool))-> (TyBool)-> (TT H. (((TyBool)-> (TyBool))-> H)-> H).l (lam w:TyBool.e)
type: ((TyBool)-> (TT M. (((TyBool)-> (TyBool))-> M)-> M))-> ((TyBool)-> (TT W. (((TyBool)-> (TyBool))-> W)-> W))-> (TyBool)-> (TT Y. (((TyBool)-> (TyBool))-> Y)-> Y)
   id:0  lam x:(TyBool)-> (TT P. (((TyBool)-> (TyBool))-> P)-> P).lam h:(TyBool)-> (TT L. (((TyBool)-> (TyBool))-> L)-> L).h
type: (TyBool)-> ((TyBool)-> (TyBool)-> (TT R. ((TyBool)-> R)-> R))-> (TyBool)-> (TT Q. ((TyBool)-> Q)-> Q)
   id:0  lam g:TyBool.lam w:(TyBool)-> (TyBool)-> (TT A. ((TyBool)-> A)-> A).w g
type: (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TT E. ((TyBool)-> E)-> E))-> (TyBool)-> (TT D. ((TyBool)-> D)-> D)
   id:0  lam s:TyBool.lam u:((TyBool)-> (TyBool))-> (TyBool)-> (TT I. ((TyBool)-> I)-> I).u (lam v:TyBool.s)
type: (TyBool)-> (TT G. (((TyBool)-> (TyBool))-> G)-> G)
   id:0  lam k:TyBool.Lam J.lam b:((TyBool)-> (TyBool))-> J.b (lam j:TyBool.k)
type: (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)
   id:1  lam f:((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool).f (lam s:TyBool.false)
   id:0  lam v:((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool).v (lam s:TyBool.true)
type: (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam r:TyBool.lam d:TyBool.d [TyBool]
type: (TyBool)-> ((TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool))-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool)
   id:0  lam o:TyBool.lam o':(TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool).o' o
type: (TyBool)-> ((TyBool)-> (TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)
   id:0  lam r:TyBool.lam u:(TyBool)-> (TyBool)-> (TyBool)-> (TyBool).u r
type: (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam s:TyBool.lam q:((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool).q (lam x:TyBool.s)
```

```
type: (TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)
   id:0  lam q:TyBool.q [(TyBool)-> (TyBool)]
type: ((TyBool)-> (TyBool)-> (TyBool))-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam f:(TyBool)-> (TyBool)-> (TyBool).lam i:(TyBool)-> (TyBool)-> (TyBool).i
type: (TyBool)-> ((TyBool)-> (TyBool))-> (TyBool)
   id:0  lam n:TyBool.lam x:(TyBool)-> (TyBool).x n
type: (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool)
   id:0  lam x:TyBool.lam o:((TyBool)-> (TyBool))-> (TyBool).o (lam w:TyBool.x)
type: (TyBool)-> (TyBool)-> (TyBool)
   id:1  lam s:TyBool.lam c:TyBool.c
   id:0  lam h:TyBool.lam u:TyBool.h
type: (TyBool)-> (TyBool)-> ((TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> ((TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (
   id:0  lam f:TyBool.lam p:TyBool.p [(TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> ((TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-
   id:0  lam j:TyBool.lam n:TyBool.n [(TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> (TyBool)-> (TyBool)-> (TyBool))-> ((TyBool)-> (TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam m:TyBool.lam h:TyBool.h [(TyBool)-> (TyBool)-> (TyBool)-> (TyBool)]
type: (TyBool)-> (TyBool)-> (((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool))-> (((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool))-> (
   id:0  lam n:TyBool.lam s:TyBool.s [((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)
   id:0  lam t:TyBool.lam f:TyBool.f [(TyBool)-> (TyBool)]
type: (TyBool)-> ((TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))->
   id:0  lam p:TyBool.lam k:(TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool).k p
type: (TyBool)-> ((TyBool)-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam b:TyBool.lam e:(TyBool)-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool).e b
type: (TyBool)-> ((TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam i:TyBool.lam m:(TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool).m i
type: (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam y:TyBool.lam n:((TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)-> (TyBool).n (lam y':TyBool.y)
type: (TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam u:TyBool.u [(TyBool)-> (TyBool)-> (TyBool)]
type: (TyBool)-> (TyBool)
   id:0  lam j:TyBool.j
type: (TyBool)-> (TyBool)-> ((TyBool)-> (TT T. ((TyBool)-> T)-> T))-> ((TyBool)-> (TT J. ((TyBool)-> J)-> J))-> (TyBool)-> (TT C. ((TyBool)-> C)-> C)
   id:0  lam g:TyBool.lam v:TyBool.v [(TyBool)-> (TT I. ((TyBool)-> I)-> I)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> (TT R. (((TyBool)-> (TyBool))-> R)-> R))-> ((TyBool)-> (TT D. (((TyBool)-> (TyBool))-> D)-> D))-> (TyBool)-> (TT D. (((TyBool)->
   id:0  lam b:TyBool.lam x:TyBool.x [(TyBool)-> (TT M. (((TyBool)-> (TyBool))-> M)-> M)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> ((TyBool)-> (TyBool))-> (TyBool))-> ((TyBool)-> ((TyBool)-> (TyBool))-> (TyBool))-> (TyBool)-> ((TyBool)-> (TyBool))-> (TyBool)
   id:0  lam d:TyBool.lam g:TyBool.g [(TyBool)-> ((TyBool)-> (TyBool))-> (TyBool)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool))-> ((TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool))-> (TyBool)-> (((TyBoo
   id:0  lam q:TyBool.lam w:TyBool.w [(TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool)]
type: (TyBool)-> (TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam c:TyBool.lam k:TyBool.k [(TyBool)-> (TyBool)-> (TyBool)]
type: (TyBool)-> ((TyBool)-> (TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))->
   id:0  lam o:TyBool.lam y:(TyBool)-> (TyBool)-> ((TyBool)-> (TyBool))-> ((TyBool)-> (TyBool))-> (TyBool)-> (TyBool).y o
type: (TyBool)-> (((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool))-> (((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool))-> ((TyBool)->
   id:0  lam g:TyBool.g [((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)]
type: ((TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool))-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool)
   id:1  lam j:(TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool).j (false)
   id:0  lam r:(TyBool)-> (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool))-> (TyBool).r (true)
type: (TyBool)-> ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)
   id:0  lam k:TyBool.lam r:(TyBool)-> (TyBool)-> (TyBool).r k
type: (TyBool)-> (((TyBool)-> (TyBool))-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)
   id:0  lam b:TyBool.lam w:((TyBool)-> (TyBool))-> (TyBool)-> (TyBool).w (lam l:TyBool.b)
type: (TyBool)-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam n:TyBool.n [TyBool]
type: ((TyBool)-> (TyBool)-> (TyBool))-> (TyBool)-> (TyBool)-> (TyBool)
   id:0  lam r:(TyBool)-> (TyBool)-> (TyBool).r
type: TyBool
   id:1  true
   id:0  false
```

D.2 EXAMPLE 1

Context:

```
/*----------------------------------------*/
gt : Int -> Int -> (TT X . X -> X -> X);
plus : Int -> Int -> Int;
minus : Int -> Int -> Int;
times : Int -> Int -> Int;
div : Int -> Int -> Int;
```

```
time : Int;
zero : Int;
one : Int;
/*********************************************/
```

The following gene pool was created randomly using the context defined above. It contains 138 elements and has a total complexity of 1135 units. The generation procedure was instructed to refuse any gene with complexity greater than 55 units.

```
type: Int-> ((Int-> Int)-> Int-> (TT Q. (Int-> Q)-> Q))-> Int-> (TT O. (Int-> O)-> O)
   id:0  lam o:Int.lam h:(Int-> Int)-> Int-> (TT H. (Int-> H)-> H).h (plus o)
type: Int
   id:34  plus time 0
   id:33  minus 0 0
   id:32  minus 1 0
   id:31  minus time 0
   id:30  plus (div time 1) 0
   id:29  plus (div time 0) 0
   id:28  plus (div time time) 0
   id:27  div (div time time) 0
   id:26  times (div time time) 0
   id:25  minus (div time time) 0
   id:24  div 1 0
   id:23  div (div time 1) 0
   id:22  times (div time 1) 0
   id:21  minus (div time 1) 0
   id:20  times 1 0
   id:19  div 0 0
   id:18  minus (times time 1) 0
   id:17  minus (times time 0) 0
   id:16  minus (times time time) 0
   id:15  minus (times time (div time time)) 0
   id:14  minus (times time (div time 0)) 0
   id:13  minus (times time (div time 1)) 0
   id:12  minus (div time 0) 0
   id:11  times time 1
   id:10  times time 0
   id:9  times time time
   id:8  times time (div time time)
   id:7  times time (div time 0)
   id:6  times time (div time 1)
   id:5  div time 1
   id:4  div time 0
   id:3  div time time
   id:2  time
   id:1  zero
   id:0  one
type: Int-> (TT P. ((TT I. I-> I-> I)-> P)-> P)
   id:0  lam e:Int.Lam C.lam l:(TT F. F-> F-> F)-> C.l (gt 0 e)
type: Int-> (TT W. W-> W-> W)
   id:3  lam l:Int.gt 0 l
   id:2  gt 1
   id:1  gt 0
   id:0  gt time
type: Int-> ((Int-> Int)-> Int-> (TT X. ((Int-> Int)-> X)-> X))-> Int-> (TT B. ((Int-> Int)-> B)-> B)
   id:0  lam r:Int.lam h:(Int-> Int)-> Int-> (TT M. ((Int-> Int)-> M)-> M).h (plus r)
type: Int-> ((Int-> Int)-> Int-> (TT R. ((TT W. W-> W-> W)-> R)-> R))-> Int-> (TT L. ((TT T. T-> T-> T)-> L)-> L)
   id:0  lam n:Int.lam s:(Int-> Int)-> Int-> (TT H. ((TT R. R-> R-> R)-> H)-> H).s (plus n)
type: Int-> Int-> Int
   id:7  lam g:Int.plus g
   id:6  lam g:Int.div g
   id:5  lam e:Int.times e
   id:4  lam q:Int.minus q
   id:3  plus
   id:2  minus
   id:1  times
   id:0  div
type: Int-> ((Int-> Int)-> Int-> Int-> Int)-> Int-> Int-> Int
   id:0  lam b:Int.lam d:(Int-> Int)-> Int-> Int-> Int.d (plus b)
type: Int-> ((Int-> Int)-> Int-> Int-> (TT J. J-> J-> J))-> Int-> Int-> (TT L. L-> L-> L)
   id:0  lam f:Int.lam p:(Int-> Int)-> Int-> Int-> (TT O. O-> O-> O).p (plus f)
type: Int-> Int
   id:58  plus (times time (div time 1))
   id:57  plus (times time (div time 0))
   id:56  plus (times time (div time time))
   id:55  plus (times time time)
   id:54  plus (times time 0)
   id:53  plus (times time 1)
   id:52  plus (minus (div time 0) 0)
   id:51  plus (minus (times time (div time 1)) 0)
```

```
   id:50  plus (minus (times time (div time 0)) 0)
   id:49  plus (minus (times time (div time time)) 0)
   id:48  plus (minus (times time time) 0)
   id:47  plus (minus (times time 0) 0)
   id:46  plus (minus (times time 1) 0)
   id:45  plus (div 0 0)
   id:44  plus (times 1 0)
   id:43  plus (minus (div time 1) 0)
   id:42  plus (times (div time 1) 0)
   id:41  plus (div (div time 1) 0)
   id:40  plus (div 1 0)
   id:39  plus (minus (div time time) 0)
   id:38  plus (times (div time time) 0)
   id:37  plus (div (div time time) 0)
   id:36  plus (plus (div time time) 0)
   id:35  plus (plus (div time 0) 0)
   id:34  plus (plus (div time 1) 0)
   id:33  plus (minus time 0)
   id:32  plus (minus 1 0)
   id:31  plus (minus 0 0)
   id:30  plus (plus time 0)
   id:29  minus (div time 0)
   id:28  minus (times time (div time 1))
   id:27  minus (times time (div time 0))
   id:26  minus (times time (div time time))
   id:25  minus (times time time)
   id:24  minus (times time 0)
   id:23  minus (times time 1)
   id:22  lam b:Int.div 0 b
   id:21  times 1
   id:20  minus (div time 1)
   id:19  times (div time 1)
   id:18  div (div time 1)
   id:17  lam b:Int.div 1 b
   id:16  minus (div time time)
   id:15  times (div time time)
   id:14  div (div time time)
   id:13  plus (div time time)
   id:12  plus (div time 0)
   id:11  plus (div time 1)
   id:10  lam e:Int.minus time e
   id:9  times time
   id:8  minus 1
   id:7  minus 0
   id:6  minus time
   id:5  plus 1
   id:4  plus 0
   id:3  plus time
   id:2  div 1
   id:1  div 0
   id:0  div time
type: Int-> ((Int-> Int)-> (TT K. K-> K-> K))-> (TT I. I-> I-> I)
   id:0  lam p:Int.lam f:(Int-> Int)-> (TT K. K-> K-> K).f (plus p)
type: Int-> Int-> (TT O. O-> O-> O)
   id:0  gt
type: Int-> (TT R. ((Int-> Int)-> R)-> R)
   id:2  lam b:Int.Lam D.lam d:(Int-> Int)-> D.d (plus b)
   id:1  lam n:Int.Lam T.lam q:(Int-> Int)-> T.q (times n)
   id:0  lam f:Int.Lam E.lam w:(Int-> Int)-> E.w (minus f)
type: Int-> ((Int-> Int)-> Int)-> Int
   id:0  lam j:Int.lam v:(Int-> Int)-> Int.v (plus j)
type: Int-> ((Int-> Int)-> Int-> Int)-> Int-> Int
   id:0  lam n:Int.lam l:(Int-> Int)-> Int-> Int.l (plus n)
type: Int-> (TT H. (Int-> H)-> H)
   id:2  lam j:Int.Lam I.lam o:Int-> I.o (div 0 j)
   id:1  lam u:Int.Lam K.lam m:Int-> K.m (div 1 u)
   id:0  lam b:Int.Lam D.lam a:Int-> D.a (minus time b)
type: Int-> ((Int-> Int)-> Int-> (TT P. P-> P-> P))-> Int-> (TT V. V-> V-> V)
   id:0  lam w:Int.lam c:(Int-> Int)-> Int-> (TT A. A-> A-> A).c (plus w)
type: TT D. D-> D-> D
   id:14  gt 0 1
   id:13  gt 0 0
   id:12  gt 0 time
   id:11  gt 0 (div time time)
   id:10  gt 0 (div time 0)
   id:9  gt 0 (div time 1)
   id:8  gt 1 1
   id:7  gt 1 0
   id:6  gt 1 time
   id:5  gt 1 (div time time)
   id:4  gt 1 (div time 0)
   id:3  gt 1 (div time 1)
   id:2  gt time 1
```

```
id:1  gt time 0
id:0  gt time time
```