Genetic Programming with Polymorphic Types and Higher-Order Functions

Franck Binard
School of Information Technology and Engineering
University of Ottawa, Canada
fbinard@site.uottawa.ca

Amy Felty
School of Information Technology and Engineering
University of Ottawa, Canada
afelty@site.uottawa.ca

ABSTRACT

This article introduces our new approach to program representation for genetic programming (GP). We replace the usual s-expression representation scheme by a strongly-typed abstraction-based representation scheme. This allows us to represent many typical computational structures by abstractions rather than by functions defined in the GP system's terminal set. The result is a generic GP system that is able to express programming structures such as recursion and data types without explicit definitions. We demonstrate the expressive power of this approach by evolving simple boolean programs without defining a set of terminals. We also evolve programs that exhibit recursive behavior without explicitly defining recursion specific syntax in the terminal set. In this article, we present our approach and experimental results.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms

Experimentation, Languages, Theory

Keywords

genetic programming, lambda calculus, polymorphism, types

1. INTRODUCTION

A Genetic Programming (GP) [11] system must be able to express a solution to the problem that it is computing. To express problem specific behavior, the system must rely on the set of primitives (terminals and functions) given to it as input. However, a GP system often needs to also express more general programming behavior. If the problem requires branching, then the system needs to be able to express branching. If the solution requires the expression of recursive behavior, then the system needs to be able to express recursion. The same is true for common data types such as lists and pairs.

We have been searching for a generic way of expressing computational structures in a GP context. We wanted our representation scheme to have the ability to express general programming behavior independently of the syntax defined in its set of primitives and we wanted it to function in a modular fashion, where computational units would be combined with other computational units to form complete programs. We also wanted to retain the ability to include problem specific constants when needed.

In [2], we presented and motivated System F [5, 14] as our representation scheme. The current paper describes a GP system that uses System F as its representation scheme and provides some preliminary results.

An extension of the simply typed λ-calculus obtained by adding an operation of abstraction on types, System F provides a general form of polymorphism. In System F programs or *terms* are best seen as computational blocks that may be plugged into each other to create new blocks [6]. Each term is associated with a *type*. The type of a program is seen as a specification of what the program does. Types also indicate when and how computational blocks may be combined. We chose System F for the following reasons:

- **Language lightness:** The pure System F syntax uses a very small set of symbols. Yet, it is possible to represent all normal programming structures (recursion, lists, trees, booleans, looping and so on) from within the language. In fact, all functions and terminals defined for GP problems in other systems that are not directly related to the problem such as branching, looping and recursion constructs can be eliminated for the same problems expressed in System F.

- **Typing structure:** System F has variables ranging over functions, data and types, making the language very expressive while still maintaining full static type-safety.

- **Strong Normalization:** System F programs always terminate. Partial normalization can be used to partially evaluate parts of programs and can be combined with abstraction support to devise a hierarchical evolution mechanism that permits the evolution not only of programs, but also of program patterns.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

In theory, we could do all our programming without going outside the pure system. This research demonstrates this by evolving binary boolean operations from empty terminal sets. In practice, genetic programs need to include primitive external functions and terminals in order to be able to express a solution to a given problem. These are defined as symbols and evaluated using a semantic evaluator that gives them meaning outside their System F representation.

This article is structured as follows: Section 2 describes System F; Section 3 summarizes related previous work; Section 4 describes our use of System F in a GP context; Section 5 describes our experiments and the results we obtained and section 6 is the concluding discussion.

2. SYSTEM F

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. The most basic combination rule is the rule that allows a function that takes an object of type A as its first argument to be combined with an object of type A. However, more complex rules might be included. 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. A classic example is an $if/then/else$ function which expresses the program: "take an argument a of type $Boolean$ and two arguments b and c as input. 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 $if/then/else$ module in a type system, it also needs to have a type. System F provides a simple and elegant solution by making it possible for functions to take types as arguments. For example, in System F, we can express $if/then/else$ by the program: "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*. An abstraction is a term that expresses behavior independently of the types of some (or all) of the arguments that it may be applied to. As another example, the System F term $(\Lambda X.\lambda x^X.\lambda y^X.x)$ 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. In this specific case, we call this term a *pure abstraction*, because there is no dependence between what the term computes and the types of its arguments. It expresses *only* general program behavior and because of this can be combined with any two computational modules of the same type. None of the nodes of the expression's parse tree (represented in figure 1) need to be labeled with primitives defined and evaluated outside the system. System F renders all the usual data-types definable as pure abstractions and directly supports recursion without naming or special operators. The expressions of the system are built using only the following operators: $\{\Pi, \Lambda, \lambda, ., \rightarrow\}$ and a mechanism for generating named variables and constant.

2.1 Types

We write $name \stackrel{\text{def}}{=} E$ when we want to use the name $name$ as a shorthand for the expression E. By convention, the names of types begin with capital letters (as in $X, Y, Int, Bool$). A type can be either a *type constant*, a *type variable* or a composition of types assembled by the following 2 rules:

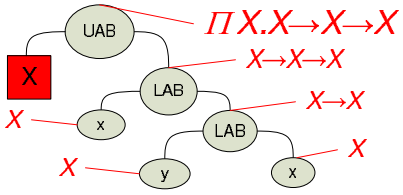

Figure 1: Parse tree for term $(\Lambda X.\lambda x^X.\lambda y^X.x)$

1. If U and V are types, then $U \rightarrow V$ is a type. $U \rightarrow V$ 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. The type $\Pi X.V$ is called an *abstract 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* 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 is right associative. 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 article, we will enclose terms 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 $Int \rightarrow Int$. In this example, the type Int is used only for the purpose of the example and is not part of System F. It is a *free type* and the pure system doesn't include (or need) any, even if for our specific purposes, we will find it simpler to include some.

The system also includes a type abstraction mechanism, the operator Π, which binds a type variable to a type expression. For example, X is bound in the type $\Pi X.X \rightarrow X \rightarrow X$. This type is the type of a function that takes a type as its argument. So if (Dec) is a term of type $\Pi X.X \rightarrow X \rightarrow X$ then the term $(Dec\ [Int])$ has type $Int \rightarrow Int \rightarrow Int$.

In terms of meaning, $\Pi X.X \rightarrow X \rightarrow X$ can be seen as the general type of all the functions that take two arguments of the same type X and evaluate to an object of type X. But it can also be seen as the type for a structure made of two constants, such as the boolean type (the set $\{true, false\}$). Before we explain how objects of a given type are constructed, we'll describe how data types are specified by the type system with some simple examples.

A structure described by a constant and a unary recursive operation (such as the natural numbers, built from a 0 constant and the successor operation) can be typed as $\Pi X.X \rightarrow (X \rightarrow X) \rightarrow X$ (or the alternate $\Pi X.(X \rightarrow X) \rightarrow X \rightarrow X$). Any structure that can be described by a constant and a recursive binary operation on an object and the structure (such as a list built from an empty list and a cons operation) can be typed by $\Pi Y.\Pi X.X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X$, so that a list of objects of type Int will have type $\Pi X.X \rightarrow (Int \rightarrow X \rightarrow X) \rightarrow X$. Structures (such as binary trees),

built from either a right and left recursive sub-structure or a leaf of a given type can be typed by the expression $\Pi Y.\Pi X.(X \to X \to X) \to (Y \to X) \to X$, so that a binary tree of Ints will have type $\Pi X.(X \to X \to X) \to (Int \to X) \to X$. The type for pairs of objects is $\Pi A.\Pi B.\Pi X.(A \to B \to X) \to X$ because a pair is built from an object of type A and an object of type B. A pair of Ints will have type $\Pi X.(Int \to Int \to X) \to X$. The type system produces a simple and elegant method to define any commonly used structure in programming without having to ever add any extra syntax. As a side note, being able to express the notion of natural numbers as pure abstraction provides the automatic ability to express the notion of iteration. And of course, recursion can be defined in terms of iteration.

As a last example, consider a location that contains either nothing or a food particle (type $Food$). Its type would be $\Pi X.X \to (Food \to X) \to X$. The next section details the process of constructing objects of a given type.

2.2 Terms

It is the terms that are the computational modules. Terms are the programs of the system when they can be evaluated further and are values, or *normal forms* when they can't. Evaluation is done by replacing all occurrences of a bound variable in a given environment. we write $\alpha[\beta/\epsilon]$ to indicate that β replaces ϵ in α. We write $y \ : \ T$ to indicate that y is a term of type T. The parse trees corresponding to the terms are built in the partial application style, which was first introduced for GP in [4] as an alternative style of program representation for typed GP. We added tags to each node to indicate the node's type and we added System F specific internal nodes as described below. Terms can be either:

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 parse tree form, these are leaves.

2. *functional applications:* If (f) is a term of type $U \to V$ and (a) is a term of type U then $(f\ a)$ is an application of type V. For applications, we omit parenthesis from left to right without loss of information and we write $(a\ b\ c\ d)$ for the term $(((a\ b)\ c)\ d)$, so $(plus\ 5\ 5)$ is really short hand for the term $((plus\ 5)\ 5)$. We represent the functional application $(f\ a)$ by a tree with root node labeled FA, with its left subtree representing the term f and its right subtree representing the term a.

3. λ-*abstractions:* Assuming that (v) is a term of type V, then $(\lambda x^U.v)$ is a term of type $U \to V$. The variable x^U is bound in (v) by the λ operator. We represent λ-abstractions in tree form by a tree with root node labeled LAB. Its left subtree is a variable and its right subtree represents a term in which the variable defined by the left subtree can appear as a leaf.

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, $(\Lambda X.\lambda p^{Int \to Int \to X}.p\ 2\ 5)$ has type $\Pi X.(Int \to Int \to X) \to X$ which we defined in 2.1 as the type for pairs of Ints. We represent this with a tree with root node labeled UAB. The left subtree is a type variable that can appear as the type of a variable defined in the right subtree by a λ-abstraction.

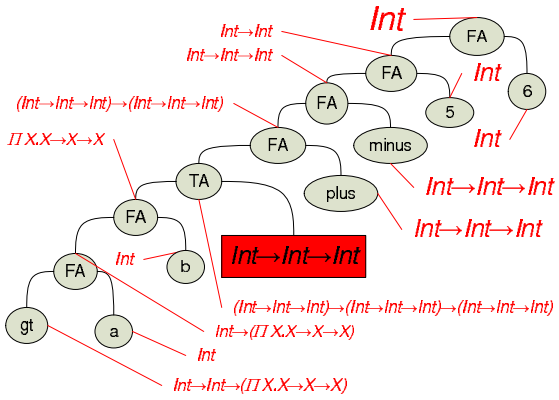

Figure 2: $(gt\ a\ b\ [Int \to Int \to Int]\ plus\ minus\ 5\ 6)$

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 $V[U/X]$. Type applications are represented by trees with root node labeled TA. The left subtree is the tree representation of a term (of abstract type) and the right subtree is a type.

2.3 Evaluation/reduction rules

Evaluation can be done in any order and always completes[9]. There are only two rules:

1. $(\lambda x.v)u$ evaluates to $v[u/x]$

2. $(\Lambda X.v)U$ evaluates to $v[U/X]$

An expression reaches its *normal form* when it can't be rewritten any further by any of the evaluation rules. It is *normalizable* if it can be reduced to a normal form and is *strongly normalizable* if every sequence of reductions starting with the expression terminates in a normal form. System F terms are strongly normalizing [9]. When we say that a term a *evaluates* to a term b, we mean that b is the normal form of a. Illustrating the first evaluation rule, the term $((\lambda x^{Int}.plus\ 5\ x)\ 4)$ evaluates to $(plus\ 5\ 4)$ and so does the term $((\lambda x^{Int \to Int \to Int}.f\ 5\ 4)\ plus)$. 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 rule to eventually evaluate to (4) (in 6 steps) while the term $(numpair\ 4\ 5\ [Int]\ (\lambda x^{Int}.\lambda y^{Int}.y))$ will evaluate to the term (5). These two terms express the projection operations of a pair. We now go back to many of the examples for which we defined types and show some of the programs associated with the types.

2.3.1 The boolean type

A boolean is one of two constants, so the type is defined as:

$$Boolean \stackrel{\text{def}}{=} \Pi X.X \to X \to X \tag{1}$$

The terms $true$ and $false$ are defined as pure abstractions and there are only two possibilities:

$$true \stackrel{\text{def}}{=} \Lambda X.\lambda x^X.\lambda y^X.x$$
$$false \stackrel{\text{def}}{=} \Lambda X.\lambda x^X.\lambda y^X.y \quad (2)$$

Having an abstract definition of the boolean type removes the need for branching-specific syntax. Boolean objects are 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 term $(gt\ a\ b\ [Int]\ 1\ 2)$ 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, $(gt\ a\ b\ [Int \rightarrow Int \rightarrow Int]\ plus\ minus\ 5\ 6)$ expresses the program "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 2.

Our last example for the boolean type is the expression of an *xor* function. This program was evolved by a GP that included no pre-defined terminals. We only used the names as definitions for the corresponding abstract expressions. We did this to make the programs human readable. The fitness function was the proportion of the number of matches in a data file that included all 4 input/output combinations encoded as described in this subsection. The solution term for xor is:

$$(\lambda a^{Boolean}.\lambda b^{Boolean}.a\ [Boolean]\ (b\ [Boolean]\ false\ true)\ b)$$

and our implementation always finds it (or another one), as well as all the other binary boolean operators.

2.3.2 Lists

The type for lists 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:

$$\Pi Y.\Pi X.X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X \quad (3)$$

and a list of objects of type U is will have type:

$$\Pi X.X \rightarrow (U \rightarrow X \rightarrow X) \rightarrow X \quad (4)$$

There are two constructors, *empty_list* and *cons*. For example, for a list of numbers, we would have the following definitions:

$$\begin{aligned} &NumList \stackrel{\text{def}}{=} \Pi X.X \rightarrow (Int \rightarrow X \rightarrow X) \rightarrow X \\ &empty\_list \stackrel{\text{def}}{=} \Lambda X.\lambda x^X.\lambda y^{(Int \rightarrow X \rightarrow X) \rightarrow X}.x \\ &cons \stackrel{\text{def}}{=} \\ &\ \lambda o^{Int}.\lambda l^{NumList}.\Lambda X.\lambda x^X.\lambda y^{(Int \rightarrow X \rightarrow X) \rightarrow X}.y\ o\ l \end{aligned} \quad (5)$$

An object of type $NumList$, for example the term $lst \stackrel{\text{def}}{=} (\Lambda X.\lambda x^X.\lambda y^{Int \rightarrow X \rightarrow X}.y\ 3\ (y\ 5\ (y\ 4\ x)))$ expresses the list of three numbers $(3 :: 5 :: 4 :: empty\_list)$. The parse tree for this term is depicted in figure 3. Once the structure for the list is built, expressing recursive behavior is very simple. For example, $(lst\ [Int]\ 0\ plus)$ is a term that expresses the program that adds all the elements of lst. It evaluates to the term $(plus\ 3\ (plus\ 5\ (plus\ 4\ 0)))$ in three steps. Using our name definitions, we could also express a list of 3 numbers by the familiar $(cons\ a\ (cons\ b\ (cons\ c\ empty\_list)))$. Note that our definition has several built in recursion schemes. For example, the following term:

$$\begin{aligned} &((cons\ a\ (cons\ b\ (cons\ c\ empty\_list)))\ [NumList] \\ &\quad empty\_list \\ &\quad (\lambda x^{Int}.\lambda l^{NumList}.cons\ (plus\ 1\ x)\ l)) \end{aligned}$$

expresses a program that takes a list as input and returns a list where every element has been incremented (this is the classic map operation). Recursion happens 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.

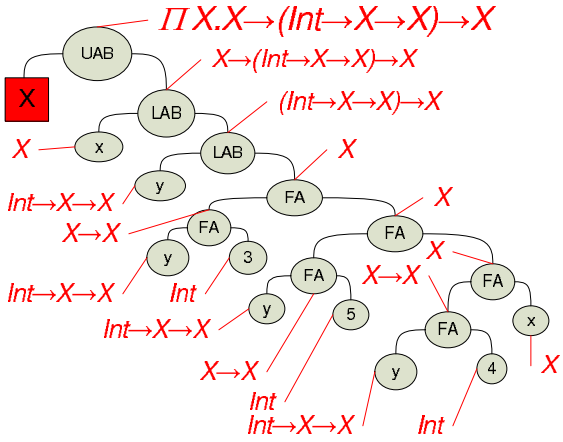

Figure 3: Parse tree for list of three integers

2.3.3 What cannot be represented in System F

System F has the strong normalization property, so the programs of the system always eventually terminate. The unsolvability of the halting problem [15] implies that there are computable functions that cannot be represented in System F. This is not so bad as it sounds because as [1] puts it, in order to find computable functions that cannot be represented in F, "one has to stand on one's head".

3. PREVIOUS RELATED WORK

In the original GP specification [11], the definitions of the primitives (the predefined terminals and functions of the system) are constrained by the *closure* requirement. All the elements in the program's parse tree must have the same type. Closure is satisfied when "any possible composition of functions and terminals produces a valid executable computer program" [12]. This implies that the programming language in which the individuals of the system are coded will always be a one-type or monomorphic language.

A mechanism called *constrained syntactic structures* is proposed in [11] to relax the closure property. Constrained syntactic structures are problem-specific syntactic rules specifying which primitives are allowed to be the child nodes of each function in the program trees. Constrained syntactic structures are used for problems requiring data typing.

In [13], Montana uses types to eliminate the closure constraint of GP systems and to restrict the search space. The

method, called *Strongly Typed Genetic Programming* (STGP) specifies a type for each argument of each function and for the value it returns. Terminals are also typed. The basic STGP formulation is equivalent to Koza's approach to constrained syntactic structures and both are limited by the need to specify multiple functions which perform the same operation. Montana resolves the issue by using *generic functions*. These are functions defined on named lists of argument types that have their return types inferred when they are embedded in a newly spawned tree. After a generic function has been instantiated (by being embedded in a new tree) it is and behaves as a standard 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 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 against the type possibility table. STGP has been applied to the problem of evolving cooperation strategies in a predator prey environment [7]. The solutions produced consistently outperformed the solutions produced by an untyped system. It is suggested that the reduced search space is the cause of the performance improvements.

The *PolyGP system* [4] is also based on a type system. Used during program creation, the type system ensures that all programs created are type-correct. PolyGP implements polymorphism using different kinds of type variables. The system that is proposed in this document differs from PolyGP in the following 4 ways:

1. In PolyGP, program parse trees are represented in a Curry style, where type information is kept distinct from the terms. This way of doing things requires the use of a type unification algorithm. We propose a Church style representation, where terms are annotated with enough type information so that there is no need for a type unification algorithm.

2. Unlike PolyGP, in the System F-based GP system, parse trees contain types, and types are evolved at the same level as terms/programs. This is the first GP system we are aware of where types are explicitly evolved during the run. This opens the door to the natural evolution of data structures and of operations on these data structures.

3. PolyGP's recursion scheme requires the definition of special structures. System F's expressive power eliminates this need.

4. PolyGP doesn't eliminate the need to predefine functions that are not directly related to the problem. A System F-based GP system does.

3.1 Representing recursive structures

There are currently two ways of providing recursion support in GP. The representation proposed in this article provides a third and new manner to achieve recursion in GP. The explicit recursion approach [3, 10], involves naming the programs of the system so that they may refer to themselves. This requires additional overhead. Each program uses a slightly different language because its name must be included in the set of instructions to which it has access. Names have to be kept and managed. Another problem with the scheme is that special mechanisms must 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. Finally, the number of recursive calls must be limited to avoid infinite loops. Each recursive call has to be tabulated while the program is running and a system of flags has to be implemented. In contrast, the recursion scheme of the System F-based system proposed in this document does not need to provide programs with the ability to call themselves in order to support recursion and has no need to check for infinite loops.

Table 1: Example context file

$Boolean = \Pi X.X \to X \to X;$
$plus : Int \to Int \to Int;$
$time : Int;$
$gt : Int \to Int \to Boolean;$

The implicit recursion approach [16] exploits PolyGP's support for higher-order functions. Recursion is implemented using predefined higher-order functions. Unfortunately, increasing the number of primitives 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. The functions are general higher-order structures with no direct relation to the actual problem for which a solution is being evolved and it is not clear how it is decided which function goes with which problem. Finally, PolyGP implements these higher-order operations on lists only. Implementing recursion on other structures requires additional syntax. The System F scheme proposed in this document doesn't share these limitations. In particular, it evolves its own higher-order functions and naturally "understands" recursion on any structure.

4. GP IMPLEMENTATION

We built an implementation using the Objective Caml language. In this section, we present the details of our use of System F in a GP context.

4.1 Problem Specification

GoalType is the type of the solution program. If the system is to evolve an *xor* function, GoalType is $Boolean \to Boolean \to Boolean$. If the system should evolve a function that takes a list of strings as argument and returns the string resulting from the concatenation of the elements of the list, then GoalType is $(\Pi X.X \to (String \to X \to X) \to X) \to String$.

4.1.1 Test cases and contexts

We pre-included the types Int, $String$ and $Float$ into the system. However, for many problems additional definitions need to be provided to the system. These are specified in the *context*. A context is a file that contains external definitions when problem specific constants or functions are required. A context might also contain names for specific System F expressions. Table 1 contains a sample context file that includes the terminals and functions that would be required for a symbolic regression system. In this case, *plus*, *time*

Table 2: Context and test cases for xor function

Context
Boolean = 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

Test cases
$false, false, false$
$false, true, true$
$true, false, true$
$true, true, false$

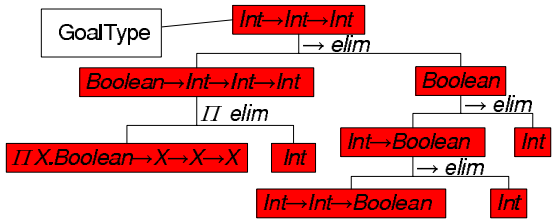

Figure 4: An example of a species

and gt are extensions, while $Boolean$ is a name for the type $\Pi X.X \to X \to X$. Problems are specified by test cases. Each test case is a comma separated list of arguments followed by the result that the function that is to be evolved should produce. For example, the evolution of an xor function is specified by the test cases listed in table 2 and (in this case) the system needs to know what $true$ and $false$ means. This meaning is provided in the context. Note that in this case, we could achieve the same results without specifying a context by using the pure System F syntax to describe the test cases. For this example, the context is nothing more than a tool to facilitate the understanding of the solution programs.

4.2 Genes

Genes are closed normalized System F terms in parse tree form. They are stored in a *gene pool* where they are grouped by types and given a unique identification key. Only one copy of a gene may exist in the gene pool at any one time. *Genotypes* are arrangements of genes, and a gene may have non-localized effects on the whole genotype. Evaluation is done at the genotype level, but selection is done at the gene level. The fitness of a genotype is (as usual) a proportionality of how close its associated program comes to describing a solution. The fitness of a gene is the average of the fitnesses of the genotypes that carry it. A gene may only exist in the gene pool if it is carried by at least one genotype in the system. For example, the context defined in table 1 would allow the definition of the gene (gt $time$ 10 $[Int \to Int \to Int]$ $plus$ $minus$) of type $Int \to Int \to Int$ which expresses the computation: "If the variable $time$ evaluates to something greater than 10 then this program evaluates to ($plus$) otherwise it evaluates to ($minus$)". Once this gene is in the gene pool, it can be used as a building block for any genotype where a gene of type $Int \to Int \to Int$ can be plugged in.

4.2.1 Complexity

There is a parsimony measure which is used to limit the size of a gene. The *complexity* of an expression is simply the number of nodes that are needed to express the gene in its parse tree form. An upper-bound on the complexity of the genes that may be included in the gene pool is provided to the system at initialization time to prevent bloat.

4.2.2 Constructing the gene pool

The gene pool is re-built at the beginning of each iteration of the system. The genes that are still in the pool after the last generation's selection process are randomly mutated to construct new genes. The construction process stops when the sum of the complexities of all objects in the pool is greater than $maxComp$, one of the system-wide variables defined for the system. The rules used to produce new terms and new types are based on logic derivation rules. Each rule is a meta-operation that takes an expression as input and outputs another expression. For example, there is a rule that starts with the input $\Pi X.X \to X \to X$, randomly picks a type Ty from the BlockPopulation set and produces the type $Ty \to Ty \to Ty$. Another rule starts with the term $t^{Ty \to Ty}$, picks a term a^{Ty} in BlockPopulation and produces the term $(t\ a)$.

4.3 Species

Once a gene pool has been generated, the genes are assembled into the working programs that will become the population of the system. The type of the programs must be GoalType in order to be compatible with the test cases. While it is a simple matter to combine genes to spawn programs, it is harder to assemble the genes into random programs of a certain type. Our solution was to use *species*. A species is a set of possible arrangements of genes that are in the gene pool. Formally, species are second-order logic proofs of GoalType that specify a schema in which any genes of an appropriate type may be plugged in. Species are represented by binary trees with nodes labeled by types with the leaves corresponding to gene pool partitions. For example, figure 4 is the tree representation of a species for a GP system that strives to evolve a function of type $Int \to Int \to Int$. Genotypes are specific arrangements of genes built on the format specified by the species. The well-known Curry-Howard isomorphism [8] applies to System F. The isomorphism provides a dual meaning to System F's type structure. For example, the type $\Pi X.X \to X \to X$ can also be read as the second-order proposition $\forall X.X \to X \to X$. Passing a type A as an argument to an object of type $\Pi X.V$ produces an object of type $V[A/X]$ which is exactly equivalent to a second-order $\forall$ elimination rule. Similarly, passing an object of type A to a function of type $A \to B$ produces an object of type B, which is exactly equivalent to a $\to$ elimination rule. In this context, including a gene of type A in the gene pool corresponds to adding the proposition A to the underlying logic system. We call a set of species paired with a gene pool an *ecosystem*.

4.4 Genotypes

A genotype is a tree. The leaves may be either pointers to genes in the gene pool or types. The roots of the subtrees may be either type applications (TA nodes) or function applications (FA nodes). For example, with the context defined in table 2 and a gene pool that includes the following genes: gene1 of type $\Pi X.Boolean \to X \to X \to X$, gene2 of type $Int \to Int \to Boolean$, gene3 and gene4 of type Int, the term ($gene1$ $[Int]$ ($gene2$ $gene3$ $gene4$))

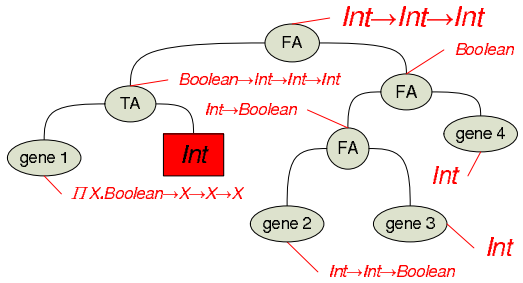

Figure 5: A genotype that belongs to the species defined by the proof in figure 4

is a valid genotype, and it belongs to the species defined by figure 4. Its tree representation is depicted in figure 5. ($gene1$ [Int] ($gene2$ $gene4$ $gene3$)) is another genotype that belongs to the same species (and in this case carries the same genes in a different arrangement). We notice and use the fact that both genotypes contain similar parts, making partial parallel evaluation possible. For example, gene4 and gene3 might each be evaluated once for both genotypes. Genotypes are ephemeral. They are all destroyed at each generation. Unless they are a solution, they only exist as statistical tools to evaluate the species and the genes. A solution is a genotype that evaluated perfectly on all test cases.

4.5 Outline of the algorithm

The user provides the following input arguments besides the context and the test cases:
$maxInds$:The desired number of genotypes per generation
$maxComp$:The approximative total size of the gene pool at each generation (measured in complexity units)
$maxSize$:The maximum size of a gene (measured in complexity units)

1. The system deduces GoalType from the test cases.
2. 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 $maxSize$ (larger genes are discarded). When the gene pool's size reaches $maxComp$, go to step 3
3. 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 $maxInds$ genotypes, go to step 5, otherwise go to step 4
4. 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 genes in the gene pool that can not be used in any of the species in the ecosystem. Go to step 2.
5. Build $maxInds$ genotypes and evaluate them on all test cases. This is highly parallelized (at the level of genotypes, we work with pointers, so genotypes can be seen as arrangement of numbers and that makes it possible to detect common sub-modules and evaluate them once for every genotypes that contain them), so it is possible to evaluate several thousand genotypes (for the problems we tested the system on) in a very short time using a regular retail computer. Score the genotypes based on the proportion of test cases they were able to compute correctly. If one or several genotypes was able to compute all test cases, then output program and terminate successfully , otherwise, go to step 6
6. Score the genes in the gene pool (as the average of the scores of the genotypes that carry them). Score the species (as the average of the scores of the genotypes that belong to them). Select (non-deterministically) genes in proportion to their fitness. Remove the unselected genes from the gene pool. Select (again, non-deterministically) species in proportion to their fitness. Remove the unselected species from the ecosystem. Kill all genotypes, go to step 2 and proceed to next generation.

5. EXPERIMENTAL RESULTS

All results below were obtained using the same implementation.

5.1 Boolean functions

Our first series of experiments was the evolution of boolean functions without the explicit definition of primitives. The context used for all operations was exactly the one defined in table 2, while the test cases were dependent on the function being evolved. Of the three parameters defined for each experience, the maximum size ($maxSize$ parameter) of the genes seemed to (counter-intuitively) be the most relevant. We were able to find a range of parameters that would guarantee that our implementation would always find (50 successes out of 50 trials) a valid solution program in less than 10 generations for any of the 8 binary boolean functions. The parameters are as follows: $maxInds \geq 1000$ (the parameter for number of genotypes per generation); $maxComp = 3000$ (the parameter for the total complexity of the gene pool) and $45 < maxSize < 75$.

Setting $maxSize$ too small (less than 35) or too large (greater than 80), however, seemed to dramatically reduce the probability of finding a solution. This is a trend we observed in all our experiments. While it is easy to see why an excessive limitation on the maximum size of the genes would prevent some essential computational blocks to be evolved, we still don't have an explanation as to why allowing larger genes to be included in the gene pool reduces the system's performance.Attempting to find an optimal value for $maxSize$, we ran 50 experiments per data point, fixing $maxInds$ at 500, $maxComp$ at 1000 and varying the value of $maxSize$ between 25 to 100 at intervals of 5 units. We found that the optimum gene size limit was 50 units (50 successes on 50 trials), with the probability of finding a solution in less that 10 generations decreasing to 0 when the value for $maxSize$ is either below 35 and above 85. The three programs below were all evolved by our system. They are the and, or and xor functions:

$or : \lambda x^{Boolean}.\lambda y^{Boolean}.x[Boolean]\ x\ y$
$and : \lambda x^{Boolean}.\lambda y^{Boolean}.x[Boolean]\ y\ (false)$
$xor : \lambda x^{Boolean}.\lambda y^{Boolean}.$
$x[Boolean]\ (y\ [Boolean]\ (false)\ x)\ y$

Table 3: Context for functions on lists of integer

```
TyList = TT X . X -> (Int->X->X)->X;
plus : Int->Int->Int;
mult : Int->Int->Int;
zero : Int ; one : Int
```

5.2 Recursion on list

For operations on list of numbers, we used the context defined in table 3, providing the test cases in pure System F. For example, $(\Lambda X.\lambda x^X.\lambda y^{X\to(Int\to X\to X)\to X}.y\ 5\ (y\ 3\ x)), 8$ is a test case for the operation that takes the list $(5 :: 3 :: empty\_list)$ as argument and returns 8. For the $empty\_list$, we used the $(\Lambda X.\lambda x^X.\lambda y^{X\to(Int\to X\to X)\to X}.x)$ representation. We ran three set of experiments, using the parameters ($maxInds$=1000, $maxComp$=3000, $maxSize$=50). The experiments and their results were:

1. Evolving a program that adds all the elements of a list. We always find one or several solutions (50 out of 50).

2. Evolving a program that multiplies all the elements of the list. We always find one or several solutions (50 out of 50).

3. Evolving a program that returns a list that contains all the elements of the input list incremented by 1. We find a solution 12 times out of 50.

The program below was evolved by our implementation using the context defined in table 3. It is a function that takes a list as arguments and returns the product of its elements.

$$\lambda x^{TyList}.x\ [Int]\ 1\ mult$$

6. CONCLUSIONS

The potential of System F as a representation scheme for GP 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 handles recursion and even allows us to define and work with many of the structures typically used by programmers. All this occurs within the system using only two abstraction operations. In terms of safety, the programs are typed and always terminate. Future work includes finding problem domains best suited to our system. More complex programs involving recursive data structures seems a natural place to start.

Acknowledgments

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada, and of Jane Street Capital, LLC for sponsoring the implementation part of the project.

7. REFERENCES

[1] H. 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.* Oxford University Press, 1992.

[2] F. Binard and A. Felty. An abstraction-based genetic programming system. In P. A. N. Bosman, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, pages 2415–2422, London, United Kingdom, 7-11 July 2007. ACM Press.

[3] S. 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.

[4] C. Clack and T. Yu. Performance enhanced genetic programming. In P. J. Angeline, R. G. Reynolds, J. R. McDonnell, and R. Eberhart, editors, *Evolutionary Programming VI*, pages 87–100, Berlin, 1997. Springer.

[5] J. 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. Fenstadt, editor, *Proc. 2nd Scandinavian Logic Symp.*, pages 63–92, Amsterdam, 1971. North-Holland.

[6] J. Girard et al. *Proofs and types*. Cambridge University Press New York, 1989.

[7] T. Haynes, R. Wainwright, S. Sen, and D. 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.

[8] W. 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.

[9] L. J.Y. Girard and Taylor. *Proofs and Types*. Cambridge University Press., 1997.

[10] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] J. R. Koza. The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In B. Soucek and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–321. John Wiley, New York, 1992.

[12] J. R. Koza. Genetic programming version 2. Submitted for inclusion in the Encyclopaedia of Computer Science and Technology, 1997.

[13] D. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3:199–230, 1995.

[14] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[15] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[16] T. Yu and C. Clack. Recursion, lambda abstractions and genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. 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.