An Abstraction-Based Genetic Programming System

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
We extend tree-based typed Genetic Programming (GP) representation schemes by introducing System F, an expressive λ-calculus, for representing programs and types. At the level of programs, System F provides higher-order programming capabilities with functions and types as first-class objects, e.g., functions can take other functions and types as parameters. At the level of types, System F provides parametric polymorphism. The expressiveness of the system provides the potential for a genetic programming system to evolve looping, recursion, lists, trees and many other typical programming structures and behavior. This is done without introducing additional external symbols in the set of predefined functions and terminals of the system. In fact, we actually remove programming structures such as $if/then/else$, which we replace by two abstraction operators. We also change the composition of parse trees so that they may directly include types.

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
Genetic Programming (GP) is an evolutionary computation search strategy in which solutions are usually represented as executable parse trees [9].

There has been some interest in experimenting with typed representations in GP. The work presented in this article exploits recent advances in type theory to extend the currently used tree-based typed GP representation schemes. Specifically, we propose to use System F [4][12], an extension of the simply typed λ-calculus obtained by adding an operation of abstraction on types, to represent the programs of a GP system. System F provides a strong form of polymorphism and can describe all functions which are provably total in second-order logic. It renders all the usual data-types definable as pure abstractions and directly supports recursion without naming or special operators. In the system, types such as $\Pi X.X \to X \to X$, the type of any function that takes two arguments of some type and returns an object of this type as its output, can be defined. A term of this type, for example $\Lambda X \lambda x^X \lambda y^X.x$, a program that takes two arguments and returns the first, will behave the same on any argument type.

System F is a worthwhile object of study as a representation scheme for GP 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.
- **Tree-like structure:** System F programs are similar in structure to the programs written in other functional languages such as Lisp. This tree-like structure is generally considered a natural choice for GP.
- **Second-order logic isomorphism:** Under the Curry-Howard isomorphism [6], System F corresponds to a second-order logic and can therefore describe all functions which are provably total in second-order logic. 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 $\forall X.X \to X \to X$. Including the atomic type A in the system 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.

- **Deriving programs from types:** There is an algorithm that takes types in a specific form as input and outputs *constructors* for the type. Constructors are functions that take the needed arguments and use these arguments to produce objects corresponding to the specified input type. For example, when given the type $\Pi X.X \rightarrow X \rightarrow X$, the algorithm produces the two terms: $\Lambda X \lambda x^X \lambda y^X.x$ and $\Lambda X \lambda x^X \lambda y^X.y$.
- **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.

This article is structured as follows: Section 2 summarizes related previous work; Section 3 describes System F and a running example expressed in this language; Section 4 describes our use of System F in a GP context; Section 5 is the concluding discussion.

2. PREVIOUS WORK

2.1 Previous work on types in GP

In the original GP specification [9], 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" [10]. 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 [9] 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 [11], 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 [5]. 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* [3] 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.

2.2 Previous work on recursion in GP

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 [8][2], 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.

The implicit recursion approach [14] 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.

3. SYSTEM F AND A RUNNING EXAMPLE

The system is formulated in the *Church style* where types are embedded in terms, resulting in an extension of the pure λ-calculus.

3.1 Rules for types

From a finite set of atomic types and an infinite set of type variables, new types are constructed as:

1. If U and V are types, then $U \rightarrow V$ is a type.
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 an object of this type can be expressed as:

$$\Lambda X.(function\_body) \tag{1}$$

Where X is 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 parameter a type ty and returns a function typed as $V[ty/X]$.

3.2 Rules for terms

1. *variables:* $x^T, y^T, z^T, \ldots$ of type T,
2. *application:* tu of type V, where t is of type $U \rightarrow V$ and u is of type U
3. *λ-abstraction:* $\lambda x^U.v$ of type $U \rightarrow V$, where x^U is a variable of type U and v is of type V,
4. *universal abstraction:* if v is a term of type V, then we can form $\Lambda X.v$ of type $\Pi X.V$, so long as the variable X is not free in the type of a free variable of v.
5. *universal application:* if t is a term of type $\Pi X.V$ and U is a type then tU is a term of type $V[U/X]$

3.3 Evaluation/reduction rules

1. $(\lambda x.v)u$ evaluates to $v[u/x]$
2. $(\Lambda X.v)U$ evaluates to $v[U/X]$

3.3.1 Naming and equivalence

We write:

$$name \overset{\text{def}}{=} E$$

to use the name *name* as a shorthand for the expression E, the two being treated as identical. The pure system is a calculus of expressions without names.

3.4 α-conversion

The α-conversion rule expresses the notion that the names of the bound variables are unimportant; for example $\lambda x^U.x$ and $\lambda y^U.y$ are the same function, and $\Pi Y.Y \rightarrow Y$ is the same type as $\Pi X.X \rightarrow X$.

3.5 Reducibility and strong normalization

An expression reaches its *normal form* when it can't be rewritten any further by 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 [7].

3.6 General scheme for free structures

The calculus is *impredicative* making self-application possible; for example, the polymorphic identity $\Lambda X.\lambda x^X.x$ can take its own type, $\Pi X.X \rightarrow X$ and then itself as arguments. The definition of an object can refer to the collection to which the object belongs. 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.

3.6.1 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. These constructors can be recursive. For example one of the list constructors creates a list from two arguments: a list of elements of a given type, and an element of the same type.

3.6.2 Format for the types

Following [7], we restrict ourselves to types in the form:

$$\Pi X.S_1 \rightarrow \ldots \rightarrow S_n \rightarrow X \tag{2}$$

with each S_i in the form:

$$T_1^i \rightarrow \ldots \rightarrow T_{k_i}^i \rightarrow X \tag{3}$$

Where the 'X' type variable in each S_i is replaced by the name of the structure. For any type ty, each $S_i[ty/X]$ is the type of a function that "builds" an object of type ty. That function is a constructor. This implies that each structure requires $f_1, \ldots, f_n$ constructor definitions. For example, if

ty is the integer type, there will be two constructors, the first one, typed as $ty \rightarrow ty$ (a constructor that takes an integer object and returns its successor, itself an integer object) and the other with type ty coding 0 (with no argument). Similarly, if ty 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.

3.6.3 The boolean type

$$ty\_Boolean \stackrel{\text{def}}{=} \Pi X.X \rightarrow X \rightarrow X \tag{4}$$

with

$$S_1 \stackrel{\text{def}}{=} X \qquad\qquad S_2 \stackrel{\text{def}}{=} X \tag{5}$$

which gives two constructors, f_1 and f_2, of type $S_1[ty\_Boolean/X]$ and $S_2[ty\_Boolean/X]$:

$$\begin{aligned} f_1 &\stackrel{\text{def}}{=} te\_true \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.x_1^X \\ f_2 &\stackrel{\text{def}}{=} te\_false \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^X.x_2^X \end{aligned} \tag{6}$$

3.6.4 Lists

The type of a list of objects of type U is definable as:

$$ty\_ListU \stackrel{\text{def}}{=} \Pi X.X \rightarrow (U \rightarrow X \rightarrow X) \rightarrow X \tag{7}$$

So there are two S_i:

$$S_1 \stackrel{\text{def}}{=} X \qquad\qquad S_2 \stackrel{\text{def}}{=} U \rightarrow X \rightarrow X \tag{8}$$

which gives two constructors, f_1 and f_2 of type ty_ListU for the empty list constructor and $U \rightarrow ty\_ListU \rightarrow ty\_ListU$ for the one that constructs an object of type ty_ListU from an object of type U and an object of type ty_ListU:

$$\begin{aligned} &f_1 \stackrel{\text{def}}{=} te\_nil \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^X.\lambda x_2^{U\rightarrow X\rightarrow X}.x_1 \\ &f_2 \stackrel{\text{def}}{=} te\_cons \stackrel{\text{def}}{=} \\ &\lambda p_1^U \lambda p_2^{ty\_ListU} \Lambda X.\lambda x_1^X.\lambda x_2^{U\rightarrow X\rightarrow X}.x_2\ p_1\ (p_2\ X\ x_1\ x_2) \end{aligned} \tag{9}$$

The second constructor 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. The normalized form of the list $(u_1, u_2, \ldots, u_n)$ of elements of type U is encoded as:

$$\Lambda X.\lambda x^X.\lambda y^{\mathrm{U}\rightarrow\ X\ \rightarrow\ X}.y\ u_1(y\ u_2(\ldots(y\ u_n\ x))\ldots) \tag{10}$$

3.6.5 Integers

The representation for integers requires two constructors: a constant (for 0) of type integer and the successor function from the integers to the integers, so:

$$ty\_Int \stackrel{\text{def}}{=} \Pi X.(X \rightarrow X) \rightarrow X \rightarrow X \tag{11}$$

And there are two S_i:

$$S_1 \stackrel{\text{def}}{=} X \rightarrow X \qquad\qquad S_2 \stackrel{\text{def}}{=} X \tag{12}$$

So we need two constructors:

$$\begin{aligned} &f_1 \stackrel{\text{def}}{=} te\_Succ \stackrel{\text{def}}{=} \lambda p_1^{ty\_Int}.\Lambda X.\lambda x_1^{X\rightarrow X}\lambda x_2^X.x_1\ (p_1\ X\ x_1\ x_2) \\ &f_2 \stackrel{\text{def}}{=} te\_0 \stackrel{\text{def}}{=} \Lambda X.\lambda x_1^{X\rightarrow X}\lambda x_2^X.x_2 \end{aligned} \tag{13}$$

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

Table 1: Primitives for GP_2

Type	
$ty\_Boolean$	Abstract Type, $\Pi X.X \rightarrow X \rightarrow X$
Int	Generic
Functions	**Type**
+	$Int \rightarrow Int \rightarrow Int$
-	$Int \rightarrow Int \rightarrow Int$
/	$Int \rightarrow Int \rightarrow Int$
*	$Int \rightarrow Int \rightarrow Int$
>	$Int \rightarrow Int \rightarrow ty\_Boolean$
Terminals	**Type**
$[0, \ldots, 10]$	Int
TIME	Int

3.6.6 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 [13] 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". 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.

3.6.7 A symbolic regression example

The problem of symbolic regression can be stated as "given a set of data points, find the underlying function in symbolic terms" [9]. Table 1 defines the primitives of GP_2, a symbolic regression GP system that uses System F to represent its individuals. GP_2 models a real valued function of the variable $TIME$.

The set of primitives is used to predefine symbols that exist outside System F and to name constructs that exist within it. The first primitives to be inserted are the atomic types. For example, the type Int. Once the atomic types have been introduced in the system, the user can insert the typed free variables of the system, such as $6 : Int$ or $+ : Int \rightarrow Int \rightarrow Int$. The set is also a syntactic sugar mechanism. Composite expressions can be included, depending on the problem. For example, the type $Int \rightarrow Int \rightarrow Int$ might be renamed as $BinaryNumericalOp$ or the booleans might be included, either by naming the type $\Pi X.X \rightarrow X \rightarrow X$ or by inserting an atomic boolean type in the set, with true and false also defined as atomic symbols. The system's designer might also decide to do both, or might even choose not to include a boolean type at all. Because it can still be expressed by the system, it is possible for it to evolve on its own.

In our sample implementation, an element of the Primitives set may also be linked to an "evaluation recipe", that provides instructions as to how the object should be evaluated. During a run, there are functions that allow the user to pause the run and extend the Primitives set so that:

1. Complex structures that appear often might be replaced by syntactic sugar, improving readability.

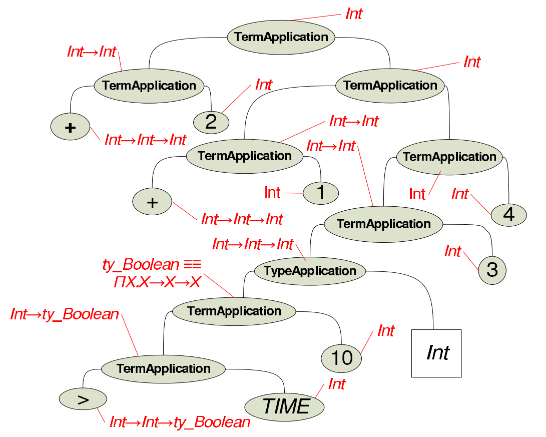

Figure 1: *Parse tree for expression 14*

2. Some System F structures that are evaluated mechanically within the system might be replaced by a more efficient evaluation recipe.

3.6.8 *Computing from within the system*

In the versions of the symbolic regression problem presented in [9] and [11], branching is enabled via the primitive function $IFTE$. GP_2 doesn't include a branching structure, and doesn't need one. For example, the expression $(+\ 2\ (+\ 1\ (IFTE\ (>\ TIME\ 10)\ 3\ 4)))$, can be expressed in System F using the primitives of GP_2 as:

$$+\ 2\ (+\ 1\ (>\ TIME\ 10\ Int\ 3\ 4\ )) \tag{14}$$

The sub-expression $(>\ TIME\ 10)$ evaluates to a $ty\_Boolean$ (as defined in section 3.6.3). $(te\_true\ Int)$ evaluates to the term $\lambda x^{Int} \lambda y^{Int}.x$ of type $Int \to Int \to Int$ after replacing X by Int. This is a function that takes two arguments of type Int as inputs and outputs the first of the arguments. Conversely, $(te\_false\ Int)$ evaluates to a function of two Int-typed arguments that returns the second of the arguments. Figure 1 represents the resulting parse tree for expression 14 in the partial application style of program representation [3]. Each node in the tree is either a leaf (corresponding to a primitive) or a typed function application that is itself at the root of a binary tree. Our research uses and augments this representation style in two ways:

1. Tags are added to each node to indicate its type.

2. Parse trees can be linked using both function application nodes and type application nodes. Type application is new to GP, and is one of the contributions of this research. Type application nodes link functions to types.

The result of the evaluation of the $(>\ TIME\ 10)$ term is directly used as a branching structure. This is only possible because true and false are functions defined without using any symbol from outside the language and therefore express abstract behavior. Boolean objects can be used as a decision procedure between two arguments of any type. For example (15) is a decision between an addition and a subtraction:

$$>\ TIME\ 10\ (Int \to Int \to Int)\ +\ - \tag{15}$$

While (16) is a simple integer-valued decision tree based on the value of the variable $TIME$:

$$>\ TIME\ 5\ Int\ (>\ TIME\ 10\ Int\ 3\ 2)\ (>\ TIME\ 0\ Int\ 1\ 0) \tag{16}$$

3.6.9 *Recursion*

As an example of recursive behavior, let's compute the Fibonacci sequence in System F using the primitives of GP_2. For this task, we may use the type:

$TrInt \overset{\text{def}}{=} \Pi X.(Int \to Int \to Int \to X) \to X$

$TrInt$ is in the required format and the constructor scheme can be applied mechanically to yield:

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

Finally let's define:

$$\begin{aligned} fib \overset{\text{def}}{=}\ & \lambda d^{TrInt}. \\ & tr\ (+\ (d\ Int\ (\lambda x^{Int} \lambda y^{Int} \lambda z^{Int}.x))\ 1) \\ & \quad (d\ Int\ (\lambda x^{Int} \lambda y^{Int} \lambda z^{Int}.z)) \\ & \quad (d\ Int\ (\lambda x^{Int} \lambda y^{Int} \lambda z^{Int}.(+\ \ y\ z))) \end{aligned}$$

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 the integers of Section 3.6.5:

$te\_n\ TrInt\ fib\ (tr\ 2\ 0\ 1)$

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

$te\_l\ TrInt\ (tr\ 2\ 0\ 1)\ (\lambda u^{ty} \lambda t^{TrInt}.fib\ t)$

where te_l is a list of n elements of any type ty in GP_2. There are many other possibilities for the computation. Just as there was no need to extend GP_2 with an $IFTE$ function for branching, there is no need to extend GP_2 for recursion.

4. SYSTEM F USE IN GP

The individuals are programs constructed as a series of applications of expressions from the BlockPopulation set, a set of well-formed normalized terms and types, built from the primitives defined for the system. BlockPopulation evolves during the system's run, but the initial set should spread out over as wide a search space area as possible. A term can be an element of the BlockPopulation set if and only if it is carried as a block by an individual of the system.

4.1 Complexity

To monitor and minimize the memory cost associated each with expression, the *complexity* of an expression is calculated as:

1. Atomic types 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 2 (one for its type variable and one for itself).

4. The complexity of a term variable is 1 plus the complexity of its type.

5. The complexity of an application term is the sum of the complexities of its two arguments and of its type, plus 1.

$$(1)\left[\lambda x^{Int\to Int}\lambda y^{Int}.xy, +(+\ 1\ 2), \left(\lambda x^{Int}\lambda y^{Int}.>\ TIME\ 10\ Int\ x\ y\right)3\ 4\right]$$

$$(2)\left[\lambda x^{Int}.+(+\ 1\ 2)(>\ x\ 10\ Int\ 3\ 4), TIME\right]$$

$$(3)\left[\lambda x^{ty\_Boolean}.+(+\ 1\ 2)(x\ Int\ 3\ 4), (>\ TIME\ 10)\right]$$

$$(4)\left[\lambda x^{Int\to Int\to Int}.x(x\ 1\ 2)(>\ TIME\ 10\ Int\ 3\ 4), +\right]$$

$$(5)\begin{bmatrix}\Lambda X\lambda x_1^{X\to X\to X}\lambda x_2^X\lambda x_3^X\lambda x_4^X\lambda x_5^X.>\ TIME\ 10\ \ (X\to X)\ \left(x_1\ x_2^X\right)\ \left(x_1\ x_3^X\right)\left(x_1\ x_4^X\ x_5^X\right), \\ Int, +, 3,\ 4,\ 1,\ 2\end{bmatrix}$$

Figure 2: *Other ways to express the phenotype produced by expression 18*

6. The complexity of an abstract term is the sum of the complexities of the body of the abstraction, of its bound term variable and of its type, plus 1.

7. The complexity of a term abstracted by type is the sum of the complexities of the body of the abstraction and of its type, plus 2.

By providing an upper-bound on the complexity of both blocks and individuals, it is possible to implement the kind of bloating controlling measures usually associated with maximum tree-depth values in typical GP systems.

4.2 Generating the initial BlockPopulation set

The Primitives set is the kernel of the BlockPopulation set. The initial set is constructed by adding new expressions, built from the ones that are already in it. The construction process stops when the sum of the complexities of all objects in the BlockPopulation set 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, a rule might start with the input $\Pi X.X \to X \to X$, randomly pick a type ty from the BlockPopulation set and produce the type $ty \to ty \to ty$. Another rule might start with the term $t^{ty\to ty}$, pick a term a^{ty} in BlockPopulation and produce the term $(t\ a)$.

4.3 The system's individuals

Genotypes are represented as lists of building blocks. The head element of an individual must be a term. For example, the genotype corresponding to (17)

$$+\ (+\ 1\ 2)\ (>\ Time\ 10\ Int\ 3\ 4) \qquad (17)$$

might be represented as either (18) or (19) in GP_2.

$$[+, +\ 1\ 2, >\ Time\ 10\ Int\ 3\ 4] \qquad (18)$$

$$[+\ (+\ 1\ 2), >\ Time\ 10\ Int\ 3\ 4] \qquad (19)$$

From a program interpretation point of view, (18) expresses its solution as the application of the function + to two Ints while (19) expresses its solution as the application of the function $f(x) = x + 3$ to a single Int. Both programs do the same thing, but have different tree structures, translating into different crossover points. Using this scheme, there are many other genotypes that can express the same phenotype. Figure 2 presents 5 other ways to express the phenotype associated with (18). Each has a unique tree structure and except for the last one, the shape of their associated tree is determined by the type of the head element.

The expressive power of System F allows the sampling of different sectors of the search space. For example, in Figure 2:

- (1) "understands" the problem as finding an operation of type $Int \to Int$ and an Int to apply it to.

- (2) sees the solution to the problem as an application of a function of type $Int \to Int$ to an Int.

- (3) looks at it from the point of view of the application of a function of type $ty\_Boolean \to Int$ to a $ty\_Boolean$.

- (4) considers the phenotype to be the application of a function of type $(Int \to Int \to Int) \to Int$ to an operator of type $Int \to Int \to Int$ such as $+$ or $-$.

- (5) is the most abstract of all (but not the most abstract possible). It sees the solution as an application of a function that takes as parameters 4 objects of some type ty and a binary operator on tys and outputs a ty object. In this sense the root function of genotype 5 is truly a polymorphic function usable with any type. Because of this, the second element of the genotype, a type, also plays a role in shaping the overall structure of the tree.

In the biological world genetically unrelated organisms will often evolve the same phenotypical solutions through completely different genetic pathways. System F's versatility produces a similar situation. In GP, it is the phenotype that is fitness-scored. This implies that (18), (19) and all the genotypes of Figure 2 will obtain the same fitness score on the same test cases. They express the same phenotype, but they are not the same at the genotype level. Because their tree structures are different, crossover will affect them in different ways.

4.3.1 GoalType

GoalType is the parameter of the system used to provide a specification of the program that is being evolved. For example, if the programs should output a list of objects of type A on input of an object of type B, then GoalType can be defined as:
$B \to (\Pi X.X \to (A \to X \to X) \to X)$
If we're asking for an operation on objects of type Int, for example as a variation of GP_2 where the $TIME$ variable is passed in as an argument to the program, then we can define GoalType as:
$Int \to Int$
and evaluate a program **Prog** as: $\mathbf{Prog}(TIME)$. GoalType does not need to be included in the Primitives, but might encourage the production of recursive behavior when it is.

4.3.2 Spawning

The first generation of individuals is randomly spawned. Randomly creating terms of type GoalType amounts (via the Curry-Howard isomorphism extended to system F) to building a proof of the proposition GoalType, using the propositions that are available as the types of the terms in the BlockPopulation set. The rules for generating individuals are simply an adaptation of second order logic derivation rules applied to the types that exist in the BlockPopulation set. Elements of Individuals are System F programs of type GoalType. At the implementation level, individuals are vectors of pointers to expressions in the BlockPopulation set.

4.4 Generalizing expressions with abstractions

The System F based representation allows the GP system to:

1. produce higher abstractions from successful specializations and
2. specialize abstractions so as to apply them in different contexts.

System F's isomorphism to second order logic makes this sort of manipulation simple because it provides a set of rules that can be used to increase or reduce the abstraction level. For example, given a term $t : U$ in which the free variable a^A occurs, we can produce the equivalent valid term $((\lambda x^A.t[x/a])\ a)$, and if it happens that A doesn't occur elsewhere than in the type of the bound variable x, then we can also produce the equivalent $((\Lambda X.\lambda x^X.t[x/a])\ A\ a)$. λ-abstraction is a way to produce generalizations. Consider now a rewrite of part of (16) into the λ-abstraction:

$$\begin{array}{l}\lambda x^{Int}\lambda y^{Int}\lambda z^{Int}. > \ TIME\ x\ Int \\ \quad (> \ TIME\ y\ Int\ 3\ 2)\ (> \ TIME\ z\ Int\ 1\ 0)\end{array} \tag{20}$$

In (20), the values of the decision cutoffs have been abstracted out of the decision tree. The System F representation supports functional abstraction independently of type abstraction.

4.4.1 Generalizing to polymorphism

To illustrate the simplicity of the process by which a polymorphic expression might be produced from a non-generic object, consider once again (16). The first step to the production of a polymorphic version of the expression is to rewrite it as a λ-abstraction. One possibility is:

$$\begin{array}{l}\lambda x^{Int}\lambda y^{Int}\lambda w^{Int}\lambda z^{Int}. > \ TIME\ 5\ Int \\ \quad (> \ TIME\ 10\ Int\ x\ y)\ (> \ TIME\ 0\ Int\ w\ z)\end{array} \tag{21}$$

Here, (21) has type $Int \to Int \to Int \to Int \to Int$ and is a function that implements the decision algorithm of (16), but takes the possible return values as input. The second step to full abstraction is to replace the type of the bound variables of the abstraction by a type variable and to add the type abstraction symbol:

$$\begin{array}{l}\Lambda Y \lambda x^{Y}\lambda y^{Y}\lambda w^{Y}\lambda z^{Y}. > \ TIME\ 5\ Y \\ \quad (> \ TIME\ 10\ Y\ x\ y)\ (> \ TIME\ 0\ Y\ w\ z)\end{array} \tag{22}$$

transforming the decision logic of the tree itself into a polymorphic function that takes 4 arguments of any type and applies the logic of (16) to the arguments. The specific advantage of the System F representation scheme in this case is the ease by which it is possible to produce a generalization of a phenotype.

4.5 The crossover operator

Crossover produces a new list of System F expressions from two parent lists in the same species. Our preliminary experiments suggest that it makes little sense to try and produce a quality offspring from two genotypes that are in completely different search space sectors. For example, in Figure 2, genotypes 5 and 3 may express the same phenotype, but any crossover operation defined on these genotypes is likely to be as absurd as a crossover operation defined on tigers and mushrooms would be. If natural species are considered solutions to a problem defined by the environment in which they exist, then it might be suggested that a species represents an optimum solution in a search landscape that is very irregular. It could be said that reproduction is a way to search for small improvements around the peak represented by the species; inter-species breeding insures that absurd search solutions are not the most common outcome of the reproduction process.

4.5.1 Species differentiator

The species differentiator that we use is *the type of the head element of its genotype list representation*. For the genotypes of Figure 2, this translates into:

- (1) belongs to the species: $(Int \to Int) \to Int \to Int$
- (2) belongs to the species $Int \to Int$
- (3) is in the species $ty\_Boolean \to Int$
- (4) is in the species $(Int \to Int \to Int) \to Int$
- (5) is in the species $\Pi X.(X \to X \to X) \to X \to X \to X \to X \to X$

It is the type of the head element that enforces the structure of the parse tree that represents the genotype.

4.5.2 Use of abstraction for crossover

The crossover scheme outlined here is specific to this system. It always produces a new program that includes everything that is common in both parents at the phenotype level and has an equal probability of including blocks that are not shared by either parent. Crossover in this system should be seen as two related but distinct operations:

- In the normal sense, crossover uses two parents that belong to the same species as material for the production of one or many children whose information is some combination of the information from the parents.
- Crossover at the genotype level also acts as both a selection and a crossover operation at the block level. When a block is identical at the same position in two parents, it becomes material for a recombination operation applied on blocks in what is in effect a selection operation on the BlockPopulation set elements. Blocks constructed from the recombination of blocks common to both parents are produced and inserted in the BlockPopulation set. In the population, these new genes are carried by the new offspring.

The general case of crossover begins by identifying the blocks that are common to both parents. When the two parents contain similar blocks, an abstraction is produced from what is similar to both programs.

Algorithm sketch for general case of crossover.
Given two parent programs within the species constraint of the crossover operation:

1. One version of each shared block is pushed onto a shared stack (SHAREDS). When two blocks at the same position in the parents are not shared, one of the two is chosen randomly and is pushed onto the argument stack (DISTINCTS). The system doesn't actually manipulates or compare actual blocks, but only the pointers that reference them inside BlockPopulation.

2. The head block of the offspring is produced as an abstraction. The expressions that are in DISTINCTS are its input and the expressions in SHAREDS are used to construct its body. The function is normalized and can be used directly as a new block by inserting it into the BlockPopulation set, where it may or may not already exist. The second step might stop here, or the head might be degraded further into two or more blocks. The blocks produced in this manner are inserted in BlockPopulation and the results of the insertion operations are used as the head blocks of the offspring.

3. The third step adds the tail blocks to the offspring's genotype by successive pop operations on DISTINCTS.

4.6 Evaluation

The evaluation procedure we use is also specific to our system. When it is possible to evaluate a block independently from the genotypes in which it is embedded, the system stores and reuses the result of the evaluation of the building block. For example, the block (+ 3 5) would be associated with its value $< 8.0 >$ at construction and would be evaluated only once. The block is replaced directly by the result of the evaluation when a genotype that contains that block is evaluated.

4.7 Sample implementation

We are currently experimenting with a sample C++ implementation of a system corresponding to GP_2 to find an optimal subset of formation rules for the generation of the BlockPopulation. Our main implementation will be produced in the functional language OCaml.

5. 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.

6. 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

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

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

[14] 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.