Verified Stack-Based Genetic Programming via Dependent Types[⋆]

Larry Diehl

http://github.com/larrytheliquid/dtgp/tree/aaip11

Abstract. Genetic Programming (GP) can act as a powerful search tool for many kinds of Inductive Programming problems. Much research has been done exploring the effectiveness of various term representations, genetic operators, and techniques for intelligently navigating the search space by taking type information into account. This paper explores the less familiar concept of formally capturing the invariants typically assumed by GP implementations. Dependently Typed Programming (DTP) extends the type-level expressiveness normally available in functional programming languages to arbitrary propositions in intuitionistic logic. We use DTP to express and enforce *semantic* invariants relevant to GP at the level of types, with a special focus on type-safe crossover for strongly typed stack-based GP. Given the complexity involved in GP implementations and the potential for introducing logic and runtime errors, we hope to help researchers avoid erroneously attributing evolutionary explanations to GP run phenomena by using a verified implementation.

1 Introduction

The goal of this work is not to come up with a novel GP algorithm with respect to evolutionary performance, but rather give an example of a non-trivial but verified and simple-to-understand GP implementation. As GP algorithms and techniques increase in complexity and sophistication, it becomes more important to verify that the parts and the whole of the algorithm are doing what is expected. Towards this end we present the groundwork of basic verified GP, with special emphasis on correctness of the crossover operation.

While the earliest work in Genetic Programming used tree structures as candidate solutions to a problem, many alternative representations have been developed since (*e.g.*, linear, graph, grammar-based). Flat linear structures are conceptually simpler than nested trees and intimately familiar to functional programmers, yet still provide competitive evolutionary results compared to tree representations [9]. As such, we will concentrate on developing a stack-based genetic programming algorithm.

Researchers concerned with formal methods have produced many different theorem provers that could be used to prove GP correctness properties. However,

[⋆] Accepted for presentation at the 4th International Workshop on Approaches and Applications of Inductive Programming.

typical GP researchers are more familiar with programming languages than proof assistants. Dependently typed languages such as Agda [6] are a nice fit because they are expressive enough at the type level to enforce invariants present in GP, while retaining the look and feel of a programming language rather than a proof assistant.

After a general overview of stack languages and dependent types, the structure of the paper will follow a common classification scheme for GP:

- **parameters:** We will start with a non-dependently typed representation, and investigate how to use standard affair dependent typing to ensure the population size parameter is adhered to.
- **representation:** We will then modify our term representation to use precise dependent types, encoding arity information in the types of candidate programs.
- **evaluation function:** We will then introduce an evaluation function for evolved terms that is assured to terminate and not otherwise diverge, by taking advantage of the host language's totality requirement.
- **genetic operators:** We will then encode the property of transitivity into the types of functions related to crossover, ensuring that ill-typed programs never enter the population.
- **initialization procedure:** Finally, we will illustrate a basic procedure to initialize our population, taking care to only randomly select programs that match the type signature of the goal program.

1.1 Stack Languages

In stack-based languages such as Forth [3] there is no distinction made between "constants" and "procedures". Instead, each syntactic element is referred to as a "word". Every word can be modeled as a function which takes the previous stack state as a value and returns the subsequent, possibly altered, state. For example, consider a small language in the boolean domain, consisting of `true`, `not`, and `and`. A word such as `true` (that would typically be considered a constant) has no requirements on the input stack, and merely returns the input stack plus a boolean value of "true" pushed on top. On the other hand, `and` requires the input stack to have at least two elements, which it pops off and evaluates before pushing their logical conjunction back onto the stack to replace them.

For monotypic languages like our example, simple typing rules emerge which assign two natural numbers to each each word. The first represents the required input stack length (the precondition), while the second represents the output stack length (the postcondition). A sequence of such words forms a stack program, for which an aggregate input/output pair exists.

During genetic operations such as crossover, stack programs must be manipulated in some manner to produce offspring for the next generation. Tchnernev [8] showed how to use arity information related to the consumed/produced stack sizes to only perform crossover at points that will produce well-typed terms. Tchnernev [9] has documented many different approaches to do this, but for simplicity of presentation we will use 1-point crossover.

1.2 Dependent Types

Dependently typed languages allow arguments in type signatures to labeled (similar to value-level variable bindings) and used elsewhere in type signature to declare *dependencies* between types and values. This paper will use the dependently typed language Agda [6] for all of its examples. [1] Agda is a purely functional language like Haskell [2], but it is distinctively total (rather than partial) and has a more expressive type system (allowing the type-checker to enforce more properties).

At compile time, Agda programs must pass two checks to prove their totality. Termination checking is accomplished by checking for structurally decreasing recursive calls. Coverage checking is accomplished by requiring that every type-correct value of a function's arguments is accounted for in the function's definition. Consequently, Agda programs do not fail to terminate [2] or crash due to unexpected input.

Thanks to totality of the language, any "value level" function can also be used in type signatures to compute more precise typing requirements (without running into undecidability of type-checking issues).

2 Parameters

For purposes of pedagogy, we will first consider how to represent a population of terms/programs in a typical non-dependent functional programming style. Thereafter, we will extend the example to use dependent types. [3]

2.1 Population List

First, let's create a new type representing the possible words to be used for some evolutionary problem.

```
data Word : Set where
  true not and : Word
```

This simple example language is intended to operate on the boolean domain using well-known constants and functions. Of course, a stack program is not merely a single word, but a sequence of them that we would like to execute in order. The familiar cons-based list can serve as a container for several words, so let us type it out.

[1] It should be possible to translate examples to similar languages such as Epigram [4] or Idris [1].

[2] Agda programs can succeed to not terminate via coinductive definition and corecursion, if controlled non-termination is what we want.

[3] For a complete and proper tutorial on dependently typed programming in Agda, see [6]

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A

Term : Set
Term = List Word
```

Notice in particular the `A : Set` part of the list type. `Set` is the type of types in Agda, and `A` is a label that acts like a variable, but at the level of type signatures. In other words, we have created a polymorphic list type which is parameterized by the kind of data it can contain. `Term` is a specific instantiation of lists that can hold the `Word`s of our example language. Below are some examples of programs we can now represent.

```
notNot : Term
notNot = not :: not :: []

anotherTrue : Term
anotherTrue = not :: not :: true :: []

nand : Term
nand = not :: and :: []
```

GP requires us to work on not one but a collection of several terms, referred to as the **population**. Normally, this might be represented as a list of lists of terms.

```
Population : Set
Population = List Term
```

While the type above is certainly functional, it leaves room for error. This brings us to our first example of preserving some GP invariant with the help of dependent types. Namely, the population that GP acts upon is expected to be a certain size, and it should stay that size as GP progresses from one generation to the next.

2.2 Population Vector

In the dependently typed world, an easy and effective way to ensure that some invariant is held is to create a type that can only possibly construct values that satisfy said invariant ("correctness-by-construction"). In our case, we would like the population size parameter to be some natural number that we specify when configuring the run. This brings us to one of the canonical examples of a dependent type, the vector. We have already seen how the list type takes a parameter to achieve polymorphism. Vectors take an additional parameter representing their length.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

Population : ℕ → Set
Population n = Vec Term n
```

The empty vector has a constant length of `zero`. The length of a vector produced by "cons" is the `suc`cessor of whatever the length of the tail is. Given such an inductive definition of a type, the natural number index of any given vector can be nothing but its length. Just like our definition of `Term`, `Population` is just a specific instantiation of a more general type (`Vec`).

As an example, here is a small population of the three terms presented earlier.

```
pop : Population 3
pop = notNot :: anotherTrue :: nand :: []
```

Once again, note that the type requires a population of exactly three terms. If we were to supply any more or less, a type error would occur at compile time. We have effectively moved checking of certain *semantic* properties of our program to compile time, meaning much less can go wrong while the program is running. [4]

Now that we have seen how to construct a dependent type, let us see how a function operating on `Vec` can make use of its properties. During selection, GP will need to retrieve a candidate program from the population. An all-too-common error (taught even in introductory level programming courses) is indexing outside the bounds of a container structure. What means do we have to prevent this from occurring? Ideally, the type of the parameter used to lookup a member should have exactly as many values as the length of our vector. This way, a bijection would exist between the lookup index type and the vector positions.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)

lookup : {A : Set} {n : ℕ} → Fin n → Vec A n → A
lookup    zero (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

The type of finite sets `Fin` has exactly `n` possible values for any `Fin n`. In the `lookup` function the natural number index is shared between the finite set and vector parameters. The effect of this sharing is that every finite set argument has exactly as many possible constructions as the length of the vector argument,

[4] In fact, the only other causes for concern are logic errors due to bad encodings by the programmer. Typical runtime errors due to non-termination or lack of coverage are disallowed by the compiler.

statically preventing any "index out of bounds" errors from occurring. Since our `Population` is merely a specific kind of vector, we are able to use the safe `lookup` when defining a function for the selection process.

3 Representation

In the previous section we represented the terms in our population as unadorned lists of words. In order to perform type-safe crossover in a manner described by [8], the type of our terms will need to be more telling.

3.1 Typing Derivation

It should come as no surprise that when we implement a type-safe version of crossover, we will need to pay close attention to the types of the terms that we are manipulating. Just as `Vec` had an extra natural number parameter for its length, we desire a `Term` type with an extra parameter for the size of the consumed/input stack, and another for the size of the produced/output stack.

Before showing a generalized list-like `Term` type for arbitrary languages, we will take a look at a more traditional embedding of a typing relation into Agda.

```
data Term (inp : ℕ) : ℕ → Set where
  []   : Term inp inp
  true : {out : ℕ} → Term inp      out   → Term m (1 + out)
  not  : {out : ℕ} → Term inp (1 + out) → Term m (1 + out)
  and  : {out : ℕ} → Term inp (2 + out) → Term m (1 + out)
```

Recall that the first parameter is the consumed stack size and the second is the produced stack size. The empty term `[]` consumes some value `inp` and produces a stack of the same size, acting as an identity program. Note also that it has no premise, so it can be considered a type-theoretical axiom.

The other three constructors are parameterized by a previous `Term` value, representing the premise of each typing rule. This `Term` representation should be understood as follows: When considering `Term 2 1` as a type alone, `2` and `3` represent the input and output stack sizes respectfully. Within the context of a constructor with a `Term` premise, the "output" position of the premise represents that word's *precondition* while the "output" position of the conclusion represent's the word's *postcondition*.

The `true` rule states that if we have some term which consumes some value `inp`, and produces another arbitrary value `out`, then the conclusion allows us to infer the existence of another term which has the same input and one additional output. In other words, `true` has a precondition that will always hold and a postcondition stating that the value in the precondition will be incremented by one.

In the `not` rule, the premise's precondition requires that the previous output be more than just any arbitrary `out`. Instead, the previous output stack size must be at least one, but can be greater. Because the `out` parameter was given in braces, Agda treats this as an implicit argument that can be unified/inferred according to other types in context. In this way, `1 + out` can represent several values such as `1 + 0` or `1 + 7`. The conclusion of `not` allows us to infer the existence of the another term of output stack size `1 + out`. This fits with our informal mental model of `not` requiring at least one argument to pop off the stack, and pushing the logical negation back on.

Finally, `and` follows the same pattern, except it requires at least two values and produces just one, leaving the output stack size exactly one less than what it was previously.

As typing derivations, our previous list-based terms look like the following (note that we have overloaded the constructors of the `Word` and `Term` types).

```
notNot : Term 1 1
notNot = not (not [])

anotherTrue : Term 0 1
anotherTrue = not (not (true []))

nand : Term 2 1
nand = not (and [])

andAnd : Term 3 1
andAnd = and (and [])
```

Our terms now have the extra consumption/production values in their type. The `andAnd` term shows how the representation correctly composes the types of several terms. The first `and` requires two values and produces one, which satisfies one of the second `and`'s requirements, resulting in a final type of `Term 3 1`.

We can highlight that the input stack remains constant throughout subterms, with an exploded view of each of the subterms in `andAnd`.

```
a : Term 3 3
a = []

b : Term 3 2
b = and a

c : Term 3 1
c = and b
```

3.2 Syntactic Non-Uniqueness

To avoid confusion, we will point out that in our representation, multiple syntactically identical terms can have different types. Specifically, what can change is the original number of arguments on the stack that the bottommost empty constructor provides.

```
empty : Term 42 42
empty = []

nand' : Term 6 5
nand' = not (and [])

andAnd' : Term 10 8
andAnd' = and (and [])
```

Being able to represent multiple different types with a syntactically identical subterm is a property that we will later exploit when defining functions to safely split and recombine terms for crossover.

3.3 Derivation Abstraction

When writing functions over term types, it would be tedious to provide a case for every word in the language. Correspondingly, we will extract the common parts among the constructors of our language into a generic `Term`, which can be thought of as abstracting out each of the typing rules presented above.

The trick is to use module parameters for the type of `Word`s, as well as *functions* for the premise/precondition and conclusion/postcondition of each rule. The result is a generic list-like `Term` structure, and has the affect of making the library not tied to any particular language to evolve.

```
module DTGP (Word : Set) (pre post : Word → ℕ → ℕ) where

data Term (inp : ℕ) : ℕ → Set where
  []  : Term inp inp
  _::_ : {n : ℕ} (w : Word) → Term inp (pre w n) → Term inp (post w n)
```

```
pre  : Word → ℕ → ℕ
pre  true  n =     n
pre  not   n = 1 + n
pre  and   n = 2 + n

post : Word → ℕ → ℕ
post true  n = 1 + n
post not   n = 1 + n
post and   n = 1 + n

open import DTGP Word pre post
```

Just like a `List` or a `Vec`, our new `Term` now only has an empty case and a cons (`_::_`) case. Now we can rewrite our examples to look just like their `List` counterparts, except with the extra useful consumption/production natural numbers in their types.

```
notNot : Term 1 1
notNot = not :: not :: []

anotherTrue : Term 0 1
anotherTrue = not :: not :: true :: []

nand : Term 2 1
nand = not :: and :: []

andAnd : Term 3 1
andAnd = and :: and :: []
```

4 Evaluation Function

When comparing relative performance between evolved terms, one typically needs to evaluate them to determine fitness . We will proceed to write an evaluation function for the example language we have used so far. Rest soundly knowing that Agda will perform a termination and coverage check to prove the totality of functions. Notice that the example below has a case for every possible term and input vector, and uses the structurally smaller tail of the input term in recursive calls.

```
eval : {inp out : ℕ} → Term inp out → Vec Bool inp → Vec Bool out
eval [] is = is
eval (true :: xs) is = true :: eval xs is
eval (false :: xs) is = false :: eval xs is
eval (not :: xs) is with eval xs is
... | o :: os = ¬ o :: os
eval (and :: xs) is with eval xs is
... | o₂ :: o₁ :: os = (o₁ ∧ o₂) :: ns
eval (or :: xs) is with eval xs is
... | o₂ :: o₁ :: os = (o₁ ∨ o₂) :: os
```

In addition to the term to evaluate, `eval` takes a vector of booleans [5] whose length `inp` is equal to the number of inputs the term expects. The return type of the function is another vector of bools `out`, matching the evaluated term's output. Both of these properties are of course enforced statically, giving more assurance that our algorithm is doing what we expect.

4.1 Fitness Function

Once again, we use a module to accept a general scoring/fitness function as a parameter. Below is an example of a function that assigns a program (which

[5] Do not be confused by the true/false constructors of the `Bool` type and `Term` types. Agda can differentiate between overloaded constructor names, according to the type they have in context.

accepts two inputs and produces one output) a score equal to the number of provided examples for which it satisfies even parity.

```
module Evolution {inp out : ℕ} (score : Term inp out → ℕ) where
```

```
score : Term 2 1 → ℕ
score xs = count (λ is → head (eval xs is) == evenParity is)
  ((true :: true :: []) :: (true :: false :: []) ::
   (false :: true :: []) :: (false :: false :: []) :: [])

open Evolution score
```

5 Genetic Operators

When writing genetic operators, *e.g.* Tchnernev's [8] 1-point crossover, we need to take subsections of different terms and recombine them in a safe manner. Tchernev points out that we need to split parent terms at a point of equal output stacks to achieve safe recombination. This leads to a question: what is the criterion for a safe append of two arbitrary terms after they have been split in this manner?

5.1 Transitive Append

Terms may have different initial input stacks, and produce different outputs according to their contained words. A safe append of two terms illustrates the transitive property.

```
_++_ : {inp mid out : ℕ} → Term mid out → Term inp mid → Term inp out
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

bc : Term 2 1
bc = and :: []

ab : Term 3 2
ab = and :: []

ac : Term 3 1
ac = bc ++ ab
```

If an attempt is made to append two terms whose input and output requirements do not satisfy one another, a compile error will occur. Using a function with a such an informative type gives a high degree of confidence that we are doing the right thing, when used inside another function such as a crossover. As we shall soon see, the type of this function in fact gives us more than simple confidence.

5.2 Transitive Split

Now that we have a function to safely recombine terms in a transitive way, we need to come up with a compatible way to split a crossover parent. In DTP a *view* [5] is a general technique for using a specialized type to reveal structural information about another type. In our case, we want to view a term as another type representing the two subsections it was split into. The following is a derivative of the `TakeView` type in [7].

```
data Split {inp out : ℕ} (mid : ℕ) :
  Term inp out → Set where
  _++'_ : (xs : Term mid out) (ys : Term inp mid) → Split B (xs ++ ys)
```

The type above captures exactly how we would like split terms to be represented, such that they can be transitively recombined. The `mid` natural number index reveals the satisfied pre/post condition point a term was split at, and the term index is the value we are splitting. The constructor carries the two subterms which share `mid` in a way that the resulting type can recombine the two via `xs ++ ys`.

Given two parent terms split in such a way, crossover needs to produce two offspring that swap the subterms at the splits. Functions for both of these swaps can be straightforwardly defined.

```
swap₁ : {inp mid out : ℕ} {xs ys : Term inp out} →
  Split mid xs → Split mid ys → Term inp out
swap₁ (xs ++' ys) (as ++' bs) = xs ++ bs

swap₂ : {inp mid out : ℕ} {xs ys : Term inp out} →
  Split mid xs → Split mid ys → Term inp out
swap₂ (xs ++' ys) (as ++' bs) = as ++ ys
```

Dependent Pairs Given some term and a natural number, we would like to split the term at an indexed position represented by the number. This function will be the key to determining the split in the female parent of a crossover. `Split` is specific enough to tell us the shared `mid` between the two subterms. However, for the purposes of this function, we do not care what `mid` is (we would actually like for the function to determine the split point for us).

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

A non-dependent pair, or tuple, carries 2 values of arbitrary types. In the dependent version of pairs, the *value* in the first component is used to determine the *type* in the second component. One common DTP technique is to use a dependent pair to hide the index type of a return value when you don't know or care what it will be. For example, sometimes we would merely like to write down a vector value and have the compiler determine the unique possible length.

```
specifiedLength : Σ ℕ (λ n → Vec Bool n)
specifiedLength = 3 , true :: false :: true :: []

discoveredLength : Σ ℕ (λ n → Vec Bool n)
discoveredLength = _ , true :: false :: true :: []
```

Note the use of an anonymous function in the type. Remember that in DTP we can do anything at the type level that we can do at the value level, including the use of the intimately-known λ. With this dependent pair trick up our sleeves, we are prepared to define `split`.

```
split : {inp out : ℕ} (n : ℕ) (xs : Term inp out) →
  Σ ℕ (λ mid → Split mid xs)
split  zero  xs = _ , [] ++' xs
split (suc n) [] = _ , [] ++' []
split (suc n) (x :: xs) with split n xs
split (suc n) (x :: ._) | _ , xs ++' ys =
  _ , (x :: xs) ++' ys
```

Because we are returning a `Split` value, the split will always hold two subterms that can be transitively combined to produce the original. In this manner, splitting `andAnd` results in two `and :: []` values of type `Term 2 1` and `Term 3 2`.

5.3 Type-Preserving Crossover

With the previous types and functions defined, defining a crossover function that takes two parent terms of the same type and returns two child terms of the same type is not far away.

Split Female For the first step in 1-point crossover we need to split the first parent (referred to here as the "female") at some random[6] point. Thus, we need to know the length of the female, then choose a random number, bounded by that length.

```
length : {inp out : ℕ} → Term inp out → ℕ
length [] = 0
length (x :: xs) = suc (length xs)

splitFemale : {inp out : ℕ} (xs : Term inp out) → ℕ →
  Σ ℕ (λ mid → Split mid xs)
splitFemale xs rand with rand mod (suc (length xs))
... | i = split (toℕ i) xs
```

[6] To keep the example as simple as possible, here we pass the random number as a parameter to the function. The final implementation uses a standard `State` monad containing a random number seed for increased modularity and to avoid mistakenly reusing a random number.

Note that we use a _mod_ function which returns a finite set representing the modulus of its two arguments. The definition of this function can be found in the supplementary source code, as it is not directly relevant to the explanation at hand.

Based upon the mid index at which the female was split, the male split can be determined by choosing a random member of all possible compatible splits.

```
splits : {inp out : ℕ} (n mid : ℕ) (xs : Term inp out) →
  Σ ℕ (λ n → Vec (Split mid xs) n)
splits zero mid xs with split zero xs
... | mid' , ys with mid =? mid'
... | yes p rewrite p = _ , ys :: []
... | no _ = _ , []
splits (suc n) mid xs with split (suc n) xs
... | mid' , ys with mid =? mid' | splits n mid xs
... | yes p | _ , yss rewrite p = _ , ys :: yss
... | no _ | _ , yss = _ , yss
```

Propositional Equality In the definition of splits, we simultaneously split at all possible positions within the male term, and filter out those possibilities that will not allow for a successful transitive recombination.

It is intuitive that the algorithm must compare the target mid of the original split to the mid' in the current split. Normally, a comparison of two terms is performed by passing them to a function that yields a boolean value, and handling the true and false cases differently. However, we need a richer version of the boolean type (the propositional equality type) whose values are associated with extra type-level information that can be used to make a Split value typecheck.

Consider the yes p case (analogous to a typical true case) within the splits zero case. We would like to return our freshly split ys value, but the type checker will not allow it. Why is this? If we look at the type signature of splits, it requires a Split mid xs, but ys is a Split mid' xs. Luckily the =? comparison function returned something more than just a boolean: it produced a constructive proof that both compared values were in fact the same. We pass the proof p (pattern matched as yes p) to Agda's rewrite keyword to convince the type checker that ys : Split mid' xs is acceptable because mid ≡ mid'.

What can we take away from all this? The primary point of interest is that the type checker requires formal constructive evidence in order to enforce invariants prescribed by the programmer. In practice, this evidence is easy to work with, as it is composed (as is everything else) of ordinary dependent types. The payoff is confidence; the burden of verifying that a program behaves as expected is lifted from the programmer's shoulders and onto the type checker's.

Split Male When we split the male parent, we choose a random member of the type-correct splits. However, this function returns a value of type Maybe, so that it may return nothing if there is no compatible split at all.

```
splitMale : {inp out : ℕ} (xs : Term inp out) →
  (mid rand : ℕ) → Maybe (Split mid xs)
splitMale xs mid rand
  with splits (length xs) mid xs
... | zero , [] = nothing
... | suc n , xss
  = just (lookup (rand mod suc n) xss)
```

Note that the proof complexity in the implementation of `splits` is isolated. Once we have a function definition that typechecks, we can freely use it without having to repeat any work.

Finally, we can write `crossover` to combine the female and male splits, and return both children using the `swaps` defined earlier.

```
crossover : {inp out : ℕ}
  (female male : Term inp out) (randF randM : ℕ) →
  Term inp out × Term inp out
crossover female male randF randM
  with splitFemale female randF
... | mid , xs with splitMale male mid randM
... | nothing = female , male
... | just ys = swap₁ xs ys , swap₂ xs ys
```

In the case where no valid male swap exists, we return the original two parents.

6 Initialization Procedure

At the onset of our GP run, we would like for our algorithm to operate on well-typed candidate programs. As such, the initialization function must be sure to only generate random type-correct programs with respect to our target program to evolve. By now, it should come as no surprise that we can (and will) enforce this requirement statically. A simple type-safe enumeration and filter strategy is adopted below.

6.1 Type-Safe Enumeration & Filter

First, we want to enumerate all terms up to some max length that conform to a given input stack size, `enum-inp`. Then, `filter-out` filters this result to include only those terms that match the desired output stack size, as well. The final list can be used as a pool to randomly select our population from.

```
enum-inp : (n inp : ℕ) → List Word → List (Σ ℕ λ out → Term inp out)
filter-out : {inp : ℕ} (out : ℕ) →
  List (Σ ℕ λ out → Term inp out) → List (Term inp out)
```

Dependent pairs are used once again, allowing us to return a list that is homogenous for `inp`, but heterogeneous for `out`. In order to implement this, we ask the user for a function that determines whether or not the precondition for a word that we want to extend a term with can be satisfied by the current output of said term.

```
module Initialization
(match : (w : Word) (out : ℕ) → Dec (Σ ℕ λ n → out ≡ pre w n))
where
```

Again, `Initialization` is another module, so the user is free to initialize the population by another means.

Decidable Relations `Dec` is a polymorphic type constructor whose values represent whether some proof of the type/proposition exists, or whether any such proof would lead to bottom ("bottom", or ⊥, is a type without constructors).

```
data Dec (P : Set) : Set where
  yes : ( p : P) → Dec P
  no  : (¬p : P → ⊥) → Dec P
```

`match` uses an existential proposition (dependent pair) inside `Dec`, and is *total* like all Agda functions. It effectively requires either a witness that the word's precondition satisfies the term's output, or a proof that no such satisfying value exists. This means that the implementor need not worry about the search for a suitable `n` ending too early, as can happen with `Maybe` (a type used commonly in this kind of situation).

```
match not zero = no ¬p where
  ¬p : Σ ℕ (λ n → 0 ≡ suc n) → ⊥
  ¬p (_ , ())
match not (suc n) = yes (n , refl)
```

The example above proves that when the output of a term is `0`, the precondition for `not` is unsatisfiable[7], and shows how to find a suitable `n` for any output greater than zero.

The definition of `enum-inp` plainly extends type-safe terms from the recursive call with the list of words argument (treating `Dec` similar to `Maybe`/partiality). `filter-out` is implemented even more straightforwardly, once again using `=?` to prove that the desired output is equal to what is returned.

[7] A pair of empty parentheses is Agda syntax used to indicate to the type checker that a value for this type is uninhabitable.

7 Conclusion

We have given an outline for a parameterized GP library whose operations are verified using dependent types. The same library can be used to evolve languages operating on domains besides booleans, such as the natural numbers, etc.

Dependent types can be used to enforce desired invariants by using informative data types and function type signatures. We have illustrated some basics for creating a verified stack-based GP implementation using type-safe 1-point crossover.

By building on a verified base, more complex GP algorithms can be created, and evolutionary data can be analyzed with much greater confidence that errors arising from implementation will not influence GP run behavior.

Hopefully, the examples presented herein can serve as a helpful template, to assist authors in encoding invariants for their particular flavors of GP within the context of dependently typed programming.

Finally, the techniques trivially extend to languages with multiple type stacks by parameterizing the main module over the domain type (e.g. $\mathbb{N} \times \mathbb{N}$), and providing a decision procedure for said type. Taking this technique further to evolve with arbitrary typing relations, rather than these Forth-like stacks, is currently under investiation.

References

1. E. C. Brady. Idris —: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM.
2. S. P. Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 2003.
3. M. G. Kelly and N. Spies. *FORTH: a text and reference.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
4. C. Mcbride. Epigram: Practical Programming with Dependent Types. pages 130–170. 2005.
5. C. Mcbride and J. Mckinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.
6. U. Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.
7. N. Oury and W. Swierstra. The power of pi. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.
8. E. Tchernev. Forth crossover is not a macromutation? 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 381–386, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
9. E. B. Tchernev. Stack-correct crossover methods in genetic programming. In E. Cantú-Paz, editor, *GECCO Late Breaking Papers*, pages 443–449. AAAI, 2002.