The Sketch Programmers Manual

June 12, 2012

1 Overview

This section provides a brief tutorial on how to run a very simple example through the compiler. The sections that follow provide detailed descriptions of all language constructs.

1.1 Hello World

To illustrate the process of sketching, we begin with the simplest sketch one can possibly write: the "hello world" of sketching.

```
harness void doubleSketch(int x){
  int t = x * ??;
  assert t == x + x;
}
```

The syntax of the code fragment above should be familiar to anyone who has programmed in C or Java. The only new feature is the symbol `??`, which is Sketch syntax to represent an unknown constant. The synthesizer will replace this symbol with a suitable constant to satisfy the programmer's requirements. In the case of this example, the programmer's requirements are stated in the form of an assertion. The keyword `harness` indicates to the synthesizer that it should find a value for `??` that satisfies the assertion for all possible inputs `x`.

Flag `-bnd-inbits` *In practice, the solver only searches a bounded space of inputs ranging from zero to* $2^{bnd\text{-}inbits}-1$. *The default for this flag is 5; attempting numbers much bigger than this is not recommended.*

1.2 Running the synthesizer

To try this sketch out on your own, place it in a file, say `test1.sk`. Then, run the synthesizer with the following command line:

```
> sketch   test1.sk
```

When you run the synthesizer in this way, the synthesized program is simply written to the console. If instead you want the synthesizer to produce standard C code, you can run with the flag `--fe-output-code`. The synthesizer can even produce a test harness for the generated code, which is useful as a sanity check to make sure the generated code is behaving correctly.

Flag `-fe-output-code` *This flag forces the code generator to produce a C++ implementation from the sketch. Without it, the synthesizer simply outputs the code to the console*

Flag `-fe-output-test` *This flag causes the synthesizer to produce a test harness to run the C++ code on a set of random inputs.* Flags can be passed to the compiler in two ways. The first and most traditional one is by passing them in the command line. For the example above, you can get code generated by invoking the compiler as follows.

```
> sketch   --fe-output-code test1.sk
```

An alternative way is to use the `pragma` construct in the language. Anywhere in the top level scope of the program, you can write the following statement:

```
pragma options " flags ";
```

This is very useful if your sketch requires a particular set of flags to synthesize. Flags passed through the command line take precedence over flags passed with `pragma`, so you can always use the command line to override options embedded in the file.

2 Core language

The core sketch language is a simple imperative language that borrows most of its syntax from Java and C.

2.1 Primitive Types

The sketch language contains only two primitive types, **`int`** and **`bit`**. The **`bit`** type is actually a subtype of **`int`**, so bit variables can be used wherever an integer is required. There are only two bit constants, `0`, and `1`. Bits are also used to represent Booleans; the bit-value `0` corresponds to **`false`**, and `1` corresponds to **`true`**.

2.2 Arrays

The SKETCH language supports fixed size arrays of any supported type. The syntax for the type constructor is as follows; if we want to declare a variable `a` to be an array of size `N` with elements of type `T`, we can declare it as:

```
T[N] a;
```

The syntax for array access is similar to that in other languages; namely, the expression `a[x]` produces an element of type `T` when the type of `a` is `T[N]`, provided that `x<N`. All array accesses are automatically checked for array bounds violations.

The constructor above works for any type `T` including other array types. This makes the semantics very simple, although it can be a little confusing for people who are used to working in languages with support for multi-dimensional arrays. To illustrate this point, consider the following example:

Example 1. *Consider the declaration below.*

```
int[N][M] a;
```

The type of `a` *is* **`int`**`[N][M]`*. This means that for an* `x < M`*,* `a[x]` *is of type* **`int`**`[N]`*, and for any* `y < N`*,* `a[x][y]` *is of type* **`int`**.

Bulk array access The indexing operation we just saw will read a single element from an array. The SKETCH language also contains support for extracting sub-arrays out of an array. If `a` is an array of type `T[N]`, we can extract a sub-array of size `M` using the following expression.

```
a[x::M]
```

If `M` is a constant greater than zero and `x + M`$\leq$`N`, then the expression `a[x::M]` produces an array of type `T[M]` containing the elements `a[x], ..., a[x+M-1]`. The expression `M` should be final, i.e. it cannot include any expressions with side effects (array accesses, function calls or unary increments) and any variables within it should be assigned only once and never changed (just like finals in Java; see more info in Section 2.3). Bulk array access of the form `a[x::M]` will generate an exception if any index between `x` and `x+M-1` is out of bounds. Specifically, the system checks that `x>=0` and `x+M <= N`, where `N` is the size of `a`. Notice that if `M` is zero, then it is legal for `x` to equal `N`.

Array assignment The language also supports bulk copy from one array to another through array assignment operator. If `a` and `b` are arrays of type `T[N]`, then the elements of `a` can be copied into `b` by using the assignment operator:

```
b = a;
```

If `a:T[N]` and `b:T[M]` are of different size, then the assignment will be legal as long as `M`$\geq$`N`. If `M`$\neq$`N`, the rhs will be padded with zeros or nulls according to the rules in Section 2.5.

Bulk array access operations can also serve as lvalues. For example, the assignment

```
b[2::4] = a[5::4]
```

is legal—assuming of course that a and b are big enough for the bulk accesses to be legal. The effect of this operation is to write values a[5], a[6], a[7], a[8] into locations b[2], b[3], b[4], b[5]. For such an assignment, the compiler will read all the values in the right hand side before writing anything to the left hand side. This is relevant when reading and writing to the same array. For example, the assignment

```
a[0::3] = a[1::3]
```

will read values `a[1], a[2], a[3]` before writing to locations `a[0], a[1], a[2]`.

Array constants Sketch supports C-style array constants. An array constant of k-elements is expressed with the following syntax.

```
{ a1, a2, ... , ak}
```

Array constants in SKETCH are more flexible than in C. They are not restricted to array initialization; they can be used anywhere an array rvalue can be used. In particular, the following are all valid statements in sketch:

```
int[3] x = {1,2,3};
x[{1,2}[a]] = 3;
x[0] = {4,5,6}[b];
x[{0,1}[a]::2] = {0,1,2,3,4,5,6}[b::2];
```

Nested array constants The entries `a1` through `ak` in the array initializer can themselves be arrays, which makes it possible for the system to support nested array initializers. The type for an array initializer will be defined by the following rule:

$$\frac{\tau = \bigsqcup \tau_i \quad \Gamma \vdash a_i : \tau_i}{\Gamma \vdash \{a_0, a_1, \ldots a_{k-1}\} : \tau[k]}$$

Given two array types $\tau_1[N]$ and $\tau_2[M]$, the type $\tau_1[N] \sqcup \tau_2[M]$ is equal to $(\tau_1 \sqcup \tau_2)[max(N, M)]$. The system pads the nested array initializers according to the rules in Section 2.5. For example, an array of the form

```
{{1,2},{1},{1,2,3},{1}}
```

will be of type int[3][4], and will be equivalent to the following array:

```
{{1,2,0},{1,0,0},{1,2,3},{1,0,0}}
```

Array Equality. The equality comparison recursively compares each element of the array and works for arrays of arbitrary types. In addition to comparing each element of the array, the equality comparison also compares the sizes of the array, so arrays of different sizes will be judged as being different even if after padding they would have been the same. In general, two arrays `a:T[n]` and `b:T[m]` will be compared according to the following recursive definition:

```
a:T[n] == b:T[m]  ⇒ n==m ∧∀ i < n a[i]==b[i]
a:T[n] == b:τ     ⇒ n==1 ∧ a[0] == b
```

In the second line, it is assumed that τ is a non-array type. There is a symmetric case when `a` is of a non-array type.

Example 2. *Given two arrays,* `int[n][m] y` *and* `int[m][n] z`*, the following assertion will always succeed:*

```
if(x==y){
        assert n==m;
}
```

That is because the only way `x` *and* `y` *can be equal is if their dimensions are equal. Similarly, given two arrays* `int[p][n][m] a` *and* `int[t] b`*, the assertion below will always succeed:*

```
if(a==b){
        assert t==m && n==1 && p == 1;
}
```

Bit Vectors While a sketch programmer can create arrays of any arbitrary type, arrays of bits allow an extended set of operations to allow programmers to easily write bit-vector algorithms. The set of allowed operators is listed below, and the semantics of each operator is the same as the equivalent operator for unsigned integers in C.

```
bit[N] & bit[M] → bit[max(N,M)]
bit[N] | bit[M] → bit[max(N,M)]
bit[N] ^ bit[M] → bit[max(N,M)]
bit[N] + bit[M] → bit[max(N,M)]
         bit[N] >> int → bit[N]
         bit[N] << int → bit[N]
                !bit[N] → bit[N]
```

Notice that most operators support operands of different sizes; the smaller array is padded to match the size of the bigger array according to the rules of padding from Section 2.5.

2.3 Dynamic Length Arrays

When you declare an array of type `T[N]`, it is possible for `N` to be an arbitrary expression. For example, consider the following code:

```
harness void main(int n, int[n] in){
 int[n] out = addone(n, in);
}
int[n] addone(int n, int[n] in){
  int[n] out;
  for(int i=0; i<n; ++i){
    out[i] = in[i]+1;
  }
  return out;
}
```

The code above illustrates one of the most common uses of dynamic length arrays: allowing functions to take arrays of arbitrary size. There are a few points worth mentioning. First, note that the size in the return array of `addone` refers to one of the parameters of the function. In general, the output type can refer to any of the input parameters, as well as to any *constant* global variables—*i.e.* global variables that are assigned a constant value upon declaration and are never changed again. Similarly, the type of an input parameter can refer to any variable that comes before it. However, output types and types of input parameters can not involve any function calls.

If a variable is used in the size of an array, its value should be final; *i.e.* the value of that variable should not change during its scope. This rules out the use of reference parameters in array length expressions. It also means that some care must be taken if the length of the array is to be computed by the function.

Example 3. *Consider a function that filters an array to return only those elements that are even. One cannot know the length of the return array a priori, because it depends on the data in the original array. One way to write such a function is as follows:*

```
int[N] filter(int N, int[N] in, ref int outsz){
        outsz = 0;
        int[N] out;
        for(int i=0; i<N; ++i){
                if(in[i]%2 == 0){
                        out[outsz++] = in[i];
                }
        }
        return out;
}
```

Notice that the function returns an array of size N, even though in reality, only the first `outsz` *elements matter. We may use the function as follows:*

```
int[N] tmp = filter(N, in, tsz);
int sz = tsz;
int[sz] filteredArray = tmp[0::sz];
```

This is admittedly awkward, and we hope to have better idioms for this in future versions of the language, but for know, that is the simplest way of producing an array of unknown length.

Flag `-bnd-arr-size` *If an input array is dynamically sized, the flag* `--bnd-arr-size` *can be used to control the maximum size arrays to be considered by the system. For any non-constant variable in the array size, the system will assume that that variable can have a maximum value of* `--bnd-arr-size`*. For example, if a sketch takes as input an array* `int[N] x`*, if* `N` *is another parameter, the system will consider arrays up to size* `bnd-arr-size`*. On the other hand, for an array parameter of type* `int[N*N] x`*, the system will consider arrays up to size* $\texttt{bnd-arr-size}^2$*.*

2.4 Structs

In addition to arrays, the SKETCH language supports heap allocated records.

To define a new record type, the programmer uses the following syntax (borrowed from C):

struct *name*{
  $type_1$ $field_1$;
  ...
  $type_k$ $field_k$;
}

One restriction on the types of fields is that you cannot have dynamic sized arrays; all array sizes must be compile time constants.

To allocate a new record in the heap, the programmer uses the keyword **new**; the syntax is the same as that for constructing an object in Java using the default constructor.

Records are manipulated through references, which behave the same way as references in Java. The following example illustrates the main properties of records and references in SKETCH.

Example 4. *The example below will behave the same way as an equivalent example would behave in Java. In particular, all the asserts will be satisfied.*

```
struct Car{
  int license;
}

void main(){
  Car c = new Car();
  Car d = c;
  c.license = 123;
  assert d.license == 123;
  strange(c, d);
  assert d.license == 123;
  assert d == c;
}

void strange(Car x, Car y){
  x = new Car();
  y = new Car();
  x.license = 456;
  y.license = 456;
  assert x.license == y.license;
  assert x != y;
}
```

Just like in Java, references are typesafe and the heap is assumed to be garbage collected (which is another way of saying the synthesizer doesn't model deallocation). A consequence of this is that a reference to a record of type T must either be **null** or point to a valid object of type T. Also, just like in Java, all pointer dereferences have an implicit null pointer check.

2.5 Automatic Padding and Typecasting

Many operations on arrays support arrays of different sizes through padding. This padding can be thought of as an implicit typecast from small arrays to bigger arrays. The objects used to pad the array depend on the type of the array. Given an array of type `T[N]`, the objects used to pad the array will be defined by the function *pad*(`T`) defined by the following rules:

pad(**int**) = 0
pad(**bit**) = 0
pad(**struct**) = **null**
pad(`T[N]`) = $\{pad(\texttt{T}), \ldots, pad(\texttt{T})\}$ *//N copies of pad(T)*

Example 5. *In the statement* `int[4] x = {1,2};`*, the right hand side has size 2, but will be implicitly cast to an array of size 4 by padding it with the value pad*(**int**)=0*, so after the assignment,* `x` *will equal* `{1,2,0,0}`*.*

A second form of implicit typecasting happens when a scalar is used in place of an array. In this case, the scalar is automatically typecast into an array of size 1.

Example 6. *Consider the following block of code*

```
struct Car{ ... }
...
Car[4] x;
Car t = new Car();
x = t;
```

This code actually involves two typecasts. First, `t` *will be typecast from the scalar type* `Car` *to the array type* `Car[1]`*. Then, the array type* `Car[1]` *will be typecast to a bigger array of type* `Car[4]` *by padding with* *pad*`(Car) = null`*. The result is that array will be equal to* `{t, null, null, null}`*.*

Example 7. *Padding also works for assignments involving nested arrays.*

```
int[2][2] x = {{2,2}, {2,2}};
int [4][4] y = x;
```

The code above involves the following implicit typecasts: first, the array x of type `int[2][2]` *is typecast into an array of type* `int[2][4]` *by padding with* *pad*`(int[2])={` *pad*`(int), pad(int)} = {0, 0}` *to produce the array* `{{2,2}, {2,2}, {0,0}, {0,0}}`*. Then, each entry in this array is typecast from* `int[2]` *to* `int[4]`*, so after the assignment, the value of* `y` *will be equal to* `{{2,2,0,0}, {2,2,0,0}, {0,0,0,0}, {0,0,0,0}}`

It is important to note that implicit casts only occur for r-values; l-values will never be implicitly typecast. In particular, this means that reference parameters to a function will never be implicitly cast and must always be of the exact size required by the signature of the callee.

2.6 Explicit Typecasting

The SKETCH language also offers some limited explicit typecasting. In particular, the language offers only two explicit typecasts:

- An array `a` of type `T[N]` can be explicitly typecast into an array of type `T[M]` by using the syntax `(T[M])a` (standard typecast notation from C). When an array is typecast to a smaller size, the remaining elements are simply truncated.
- A bit array `bit[N]` can be explicitly typecast into an integer. When this happens, the first bit in the array is interpreted as the least significant bit and the last one as the most significant bit. The reverse cast from an integer to a bit array is not supported.

Example 8. *One instance where explicit casting is useful is when comparing an array against the zero array.*

```
int[N] x=...;
assert x == (bit[N])0;
```

Notice that in the code above, if we had written simply `x==0` *in the assertion, the assertion would have been violated when* `N>1`*, because the scalar zero is treated as an array of size 1. By casting the constant zero into an array of size* `N`*, we ensure that x is compared against an array of size* `N` *consisting of all zeros.*

Example 9. *Explicit casting is also useful when copying one dynamically sized array into another one.*

```
int[N] x=...;
int[M] y = (bit[M])x;
```

If we knew that `N` *is smaller than* `M`*, we could have written simply* `y=x`*, and the automatic padding would have made the assignment correct. Similarly, if we knew that* `M` *is smaller than or equal to* `N`*, assigning* `y=x[0::M]` *would have been legal. However,* `y=x` *fails when* `M` *is smaller than* `N`*, and x[0::M] fails when* `M>N`*. The cast on the other hand succeeds in both cases and has the expected behavior.*

2.7 Control Flow

The language supports the following constructs for control flow: **if-then**, **while**, **do-while**, **for**. These have the same syntax and semantics as in C/C++ or Java. The language does not have a **switch** statement, although it is likely to be added in a future version of the language. The language also does not support **continue** and **break**, although they can easily be emulated with **return** by using closures (see Section 2.11).

The synthesizer reasons about loops by unrolling them. The degree of unrolling is controlled by a flag `--bnd-unroll-amnt`. If the loop iteration bounds are static, however, the loop will be unrolled as many times as necessary to satisfy the static bounds.

Flag `-bnd-unroll-amnt` *This flag controls the degree of unrolling for both loops and* **repeat** *constructs*

Example 10. *Consider the three loops below.*

```
for(int i=0; i<N; ++i){...}
for(int i=0; i<100; ++i){...}
for(int i=0; i<N && i<7; ++i){...}
```

If `N` *is an input variable, the first loop will be unrolled as many times as specified by* `--bnd-unroll-amnt`*. The second loop will be unrolled* `100` *times regardless of the value of the flag. For the third loop, the unroll factor will be controlled by the flag, but will never exceed seven.*

2.8 Global variables

The sketch language supports global variables with one important restriction: global arrays must be of constant dimension. You can use global variables for the dimension of an array as long as the global variable is constant. Global variables that are passed as reference parameters, or that are passed as parameters to function parameters will be considered to be non-constant.

2.9 Functions

The sketch language also supports functions. The syntax for declaring a function is the same as in C.

```
ret_type name(args){
  body
}
```

Recursion The synthesizer reasons about function calls by inlining them into their calling context. In principle, this could be problematic for recursive functions, but in practice this usually is not a problem. The synthesizer uses a flag `bnd-inline-amnt` to bound the maximum number of times a function can be inlined. If any input requires inlining more than the allowed number of times, synthesis will fail.

Flag `-bnd-inline-amnt` *Bounds the amount of inlining for any function call. The value of this parameter corresponds to the maximum number of times any function can appear in the stack.*

Reference Vs. Value Parameter Passing By default, parameter passing is done by value; however, it is possible to pass parameters by reference by prefixing them with the keyword **ref**.

Only local variables should ever be passed by reference, and reference parameters should never be aliased. The reason for this restriction is that the synthesizer models reference parameters using copy-in-copy-out semantics. If the parameters are local variables and are not aliased, then copy-in-copy-out is indistinguishable from pass-by-reference.

2.10 Function parameters

Functions can also take functions as parameters. We use the keyword `fun` to denote a function type. The example below illustrates the use of function parameters.

```
int apply(fun f, int x){
        return f(x);
}
int timesTwo(int x){
        return x+x;
}

harness void main(int x){
        assert apply(timesTwo, x) == 2*x;
}
```

The language imposes several restrictions on the use of the `fun` type. First, the type can only be used for parameters. You cannot declare a variable or a data-structure field of type `fun`. There are also no operators defined for functions; in particular, the ternary operator `?:` cannot be used with functions. You can also not create arrays of functions, and you cannot use functions as return values or reference parameters to a function. In short, functions are not quite first class citizens in sketch, but function parameters do enable some very useful idioms.

2.11 Local functions and closures

Sketch supports the definition of functions inside other functions. The syntax for doing this is the same as when the function is defined outside a function. The body of the locally defined function can access any variable that is in scope in the context of the function definition. The example below illustrates how local functions can be used together with high-order functions.

```
void ForLoop(fun f, int i, int N){
        if(i<N){
                f(i);
                ForLoop(f, i+1, N);
        }
}

harness void main(int N, int[N] A){
        int[N] B;
        void copy(int i){
                B[i] = A[i];
        }
        ForLoop(copy, 0, N);
        assert A == B;
}
```

In the sketch above, `ForLoop` takes the closure involving the function `copy` and its local environment; the effect of the call to `ForLoop` is the same as if the body of copy had been placed in a traditional **`for`** loop.

2.12 Packages

The SKETCH language supports packages. A package is identified by the `package` statement at the beginning of a file.

```
package PACKAGENAME;
```

All the functions and structures defined in a file must belong to the same package, so the compiler will produce an error if there is more than one package definition in a file. If a file does not have a `package` command, then by default its contents will belong to the package `ANONIMOUS`. Also, note that unlike Java, package names cannot have periods or other special symbols.

A file can import other packages by using the `include` command. The syntax of the command is shown below. The string in quotes corresponds to the name of the file where the package resides.

```
include "file.sk";
```

The include command should not be confused with the `#include` preprocessor directive, which simply inlines the contents of a file and is not really part of the language.

Flag `-fe-inc` *The command line flag –fe-inc can be used to tell the compiler what directories to search when looking for included packages. The flag works much like the* `-I` *flag in gcc, and can be used multiple times to list several different directories.*

Each package defines its own namespace, allowing the system to avoid name conflicts. Code in one package can explicitly refer to functions or structures defined in another package by using the `@` notation. For example, a call of the form `foo@pk()` will call a function `foo` defined in package `pk`. Similarly, a declaration of the form `Car@vehicles c = new Car@vehicles()` defines a new object of type `Car`, where the type was defined in the package `vehicles`. In the absence of an explicit package name, the system will search for definitions of functions and structures as follows:

- If the name is defined locally in the same package, the local definition will be used.
- If the name is not defined locally in the same package, but is only defined in one other package (so there is no ambiguity), then the definition in that other package will be used.
- If the name is not defined locally in the same package and the same name is defined in multiple packages, then you need to explicitly name the package or you will get a compiler error.

Example 11. *The example below illustrates the use of packages.*

```
// Begin file farm.sk
package farm;
struct Goat{
   int weight;  }
struct Ram{
   int age;     }
struct Mouse{
   int age;     }
// End file farm.sk
```

```
// Begin file computer.sk
package computer;
struct Cpu{
   int freq;    }
struct Ram{
   int size;    }
struct Mouse{
   bit isWireless; }
// End file computer.sk
```

```
//Begin file test.sk
include "computer.sk";
include "farm.sk"
struct Mouse{
   int t;
}
harness main(){
   Cpu c = new Cpu(); // No ambiguity here.
   Ram@farm r = new Ram@farm() //Without @farm, this would be an error.
   Ram@computer rc = new Ram@computer();
   Mouse m = new Mouse(); // Give preference to the locally defined mouse.
   m.t = 10;
}
//End file test.sk
```

3 Constant Generators and Specs

Sketching extends a simple procedural language with the ability to leave *holes* in place of code fragments that are to be derived by the synthesizer. Each hole is marked by a generator which defines the set of code fragments that can be used to fill a hole. SKETCH offers a rich set of constructs to define generators, but all of these constructs can be described as syntactic sugar over a simple core language that contains only one kind of generator: an unknown integer constant denoted by the token **??**.

From the point of view of the programmer, the integer generator is a placeholder that the synthesizer must replace with a suitable integer constant. The synthesizer ensures that the resulting code will avoid any assertion failures under any input in the input space under consideration. For example, the following code snippet can be regarded as the "Hello World" of sketching.

```
harness void main(int x){
   int y = x * ??;
   assert y == x + x;
}
```

This program illustrates the basic structure of a sketch. It contains three elements you are likely to find in every sketch: (i) a **harness** procedure, (ii) holes marked by generators, and (iii) assertions.

The harness procedure is the entry point of the sketch, and together with the assertion it serves as an operational specification for the desired program. The goal of the synthesizer is to derive an integer constant C such that when **??** is replaced by C, the resulting program will satisfy the assertion for all inputs under consideration by the verifier. For the sketch above, the synthesized code will look like this.

```
void main(int x){
   int y = x * 2;
   assert y == x + x;
}
```

3.1 Types for Constant Generators

The constant hole **??** can actually stand for any of the following different types of constants:

- Integers (**int**)
- Booleans (**bit**)
- Constant sized arrays and nested constant sized arrays

The system will use a simple form of type inference to determine the exact type of a given hole.

3.2 Ranges for holes

When searching for the value of a constant hole, the synthesizer will only search values greater than or equal to zero and less than 2^N, where N is a parameter given by the flag `--bnd-ctrlbits`. If you wan to be explicit about the number of bits for a given hole, you can state it as **??(N)**, where `N` is an integer constant.

Flag `-bnd-ctrlbits` *The flag* `bnd-ctrlbits` *tells the synthesizer what range of values to consider for all integer holes. If one wants a given integer hole to span a different range of values, one can use the extended notation* **??(N)***, where* `N` *is the number of bits to use for that hole.*

3.3 Generator functions

A generator describes a space of possible code fragments that can be used to fill a hole. The constant generator we have seen so far corresponds to the simplest such space of code fragments: the space of integers in a particular range. More complex generators can be created by composing simple generators into *generator functions*.

As a simple example, consider the problem of specifying the set of linear functions of two parameters `x` and `y`. That space of functions can be described with the following simple generator function:

```
generator int legen(int i, int j){
   return ??*i + ??*j+??;
}
```

The generator function can be used anywhere in the code in the same way a function would, but the semantics of generators are different from functions. In particular, every call to the generator will be replaced by a concrete piece of code in the space of code fragments defined by the generator. Different calls to the generator function can produce different code fragments. For example, consider the following use of the generator.

```
harness void main(int x, int y){

  assert legen(x, y) == 2*x + 3;
  assert legen(x,y) == 3*x + 2*y;

}
```

Calling the solver on the above code produces the following output

```
void _main (int x, int y){
  assert ((((2 * x) + (0 * y)) + 3) == ((2 * x) + 3));
  assert (((3 * x) + (2 * y)) == ((3 * x) + (2 * y)));
}
```

Note that each invocation of the generator function was replaced by a concrete code fragment in the space of code fragments defined by the generator.

The behavior of generator functions is very different from standard functions. If a standard function has generators inside it, those generators are resolved to produce code that will behave correctly in all the calling contexts of the function as illustrated by the example below.

```
int linexp(int x, int y){
   return ??*x + ??*y + ??;
}
harness void main(int x, int y){
   assert linexp(x,y) >= 2*x + y;
   assert linexp(x,y) <= 2*x + y+2;
}
```

For the routines above, there are many different solutions for the holes in `linexp` that will satisfy the first assertion, and there are many that will satisfy the second assertion, but the synthesizer will chose one of the candidates that satisfy them both and produce the code shown below. Note that the compiler always replaces return values for reference parameters, but other than that, the code below is what you would expect.

```
void linexp (int x, int y, ref int _out){
  _out = 0;
  _out = (2 * x) + (1 * y);
  return;
}
```

```
void _main (int x, int y){
  int _out = 0;
  linexp(x, y, _out);
  assert (_out >= ((2 * x) + y));
  int _out_0 = 0;
  linexp(x, y, _out_0);
  assert (_out_0 <= (((2 * x) + y) + 2));
}
```

3.4 Recursive Generator Functions

Generators derive much of their expressive power from their ability to recursively define a space of expressions.

```
generator int rec(int x, int y, int z){
   int t = ??;
   if(t == 0){return x;}
   if(t == 1){return y;}
   if(t == 2){return z;}

   int a = rec(x,y,z);
   int b = rec(x,y,z);

   if(t == 3){return a * b;}
   if(t == 4){return a + b;}
   if(t == 5){return a - b;}
}

harness void sketch( int x, int y, int z ){
   assert rec(x,y, z) == (x + x) * (y - z);
}
```

3.5 Regular Expression Generators

Sketch provides some shorthand to make it easy to express simple sets of expressions. This shorthand is based on regular expressions. Regular expression generators describe to the synthesizer a set of choices from which to choose in searching for a correct solution to the sketch. The basic syntax is

| regexp |

Where the regexp can use the operator | to describe choices, and the operator ? to define optional subexpressions.

For example, the sketch from the previous subsections can be made more succinct by using the regular expression shorthand.

```
generator int rec(int x, int y, int z){
    if(??){
        return {| x | y | z |};
    }else{
        return {| rec(x,y,z) (+ | - | *) rec(x,y,z)  |};
    }
}

harness void sketch( int x, int y, int z ){
   assert rec(x,y, z) == (x + x) * (y - z);
```

```
}
```

Regular expression holes can also be used with pointer expressions. For example, suppose you want to create a method to push a value into a stack, represented as a linked list. You could sketch the method with the following code:

```
push(Stack s, int val){
  Node n = new Node();
  n.val = val;
  {|  (s.head | n)(.next)? |} =   {|  (s.head | n)(.next)? |};
  {|  (s.head | n)(.next)? |} =   {|  (s.head | n)(.next)? |};
}
```

3.6 High order generators

Generators can take other generators as parameters, and they can be passed as parameters to either generators or functions. This can be very useful in defining very flexible classes of generators. For example, the generator rec above assumes that you want expressions involving three integer variables, but in some cases you may only want two variables, or you may want five variables. The following code describes a more flexible generator:

```
generator int rec(fun choices){
    if(??){
        return choices();
    }else{
        return {| rec(choices) (+ | - | *) rec(choices)  |};
    }
}
```

We can use this generator in the context of the previous example as follows:

```
harness void sketch( int x, int y, int z ){
   generator int F(){
        return {| x | y | z |};
   }
   assert rec(F) == (x + x) * (y - z);
}
```

In a different context, we may want an expression involving some very specific sub-expressions, but the same generator can be reused in the new context.

```
harness void sketch( int N, int[N] A, int x, int y ){
   generator int F(){
        return {| A[x] | x | y |};
   }
   if(x<N){
           assert rec(F) == (A[x]+y)*x;
   }
}
```

High order generators can also be used to describe patterns in the expected structure of the desired code. For example, if we believe the resulting code will have a repeating structure, we can express this with the following high-order generator:

```
generator void rep(int n, fun f){
    if(n>0){
```

```
        f();
        rep(n-1, f);
    }
}
```

4 Glossary of Flags

This is a glossary of flags

-bnd-arr-size If an input array is dynamically sized, the flag `--bnd-arr-size` can be used to control the maximum size arrays to be considered by the system. For any non-constant variable in the array size, the system will assume that that variable can have a maximum value of `--bnd-arr-size`. For example, if a sketch takes as input an array **`int`**`[N] x`, if `N` is another parameter, the system will consider arrays up to size `bnd-arr-size`. On the other hand, for an array parameter of type **`int`**`[N*N] x`, the system will consider arrays up to size $\texttt{bnd-arr-size}^2$.. 5

-bnd-ctrlbits The flag `bnd-ctrlbits` tells the synthesizer what range of values to consider for all integer holes. If one wants a given integer hole to span a different range of values, one can use the extended notation **`??`**`(N)`, where `N` is the number of bits to use for that hole.. 11

-bnd-inbits In practice, the solver only searches a bounded space of inputs ranging from zero to $2^{\texttt{bnd-inbits}} - 1$. The default for this flag is 5; attempting numbers much bigger than this is not recommended.. 1

-bnd-inline-amnt Bounds the amount of inlining for any function call. The value of this parameter corresponds to the maximum number of times any function can appear in the stack.. 8

-bnd-unroll-amnt This flag controls the degree of unrolling for both loops and **`repeat`** constructs. 8

-fe-inc The command line flag –fe-inc can be used to tell the compiler what directories to search when looking for included packages. The flag works much like the `-I` flag in gcc, and can be used multiple times to list several different directories.. 10

-fe-output-code This flag forces the code generator to produce a C++ implementation from the sketch. Without it, the synthesizer simply outputs the code to the console. 1

-fe-output-test This flag causes the synthesizer to produce a test harness to run the C++ code on a set of random inputs.. 2