The Sketch Programmers Manual

Armando Solar-Lezama

For Sketch Version 1.7.6

Contents

1 Overview **3**
1.1 Hello World . . . 3
1.2 Running the synthesizer . . . 3
1.3 Parallel Solving . . . 4

2 Core language **4**
2.1 Primitive Types . . . 4
2.2 Structs . . . 4
2.3 Temporary Structures . . . 6
2.4 Final Types . . . 7
2.5 Arrays . . . 7
2.6 Automatic Padding and Typecasting . . . 12
2.7 Explicit Typecasting . . . 13
2.8 Algebraic Data types . . . 14
2.9 Struct inheritance . . . 16
2.10 Control Flow . . . 16
2.11 Functions . . . 17
2.12 Function parameters . . . 19
2.13 Local functions and closures . . . 20
2.14 Lambda Functions . . . 21
2.15 Polymorphism . . . 22
2.16 Uninterpreted Functions . . . 24
2.17 Packages . . . 24
2.18 Global variables . . . 25
2.19 Annotation System . . . 26
2.20 The `@Native` annotation . . . 26
2.21 Casting of Expressions . . . 28

3 Constant Generators and Specs **28**
3.1 Harnesses and function equivalence requirement . . . 29
3.2 Assumptions . . . 30
3.3 Types for Constant Generators . . . 30
3.4 Ranges for holes . . . 31
3.5 Minimizing Hole Values . . . 31

4 Generator functions **32**
4.1 Recursive Generator Functions . . . 33
4.2 Regular Expression Generators . . . 35
4.3 Local Variables Construct . . . 36
4.4 High order generators . . . 36
4.5 `repeat` construct . . . 37

5 Regression tests and Benchmark Suite **38**

6 Advanced Usage and Diagnostics **39**
6.1 Interpreting Synthesizer Output . . . 39
6.2 Parallel Solving . . . 39
6.3 Performance diagnostics . . . 39
6.4 Custom Code Generators . . . 42
6.5 Temporary Files and Frontend Backend Communication . . . 42
6.6 Extracting the intermediate representation . . . 43

7 Standard Library **44**
7.1 Package math . . . 44
7.1.1 Functions . . . 44
7.2 Package generics . . . 46
7.2.1 Functions . . . 46
7.3 Package generators . . . 47
7.3.1 Functions . . . 47
7.4 Package array . . . 48
7.4.1 Datastructures . . . 48
7.4.2 Functions . . . 48
7.5 Package list . . . 49
7.5.1 Datastructures . . . 49
7.5.2 Functions . . . 49
7.6 Package stack . . . 50
7.6.1 Datastructures . . . 50
7.6.2 Functions . . . 50

8 Credits **50**

9 Glossary of Flags **51**

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.

1.3 Parallel Solving

The SKETCH synthesizer has a parallel mode which can yield significant speedups on certain problems. Parallel mode can be invoked by running with the flag `--slv-parallel`. By default, this flag will use one less than the total number of processors available in your system, but you can control the exact number of processors used by passing the additional flag `--slv-p-cpus`. Not all problems will benefit from parallel solving; some problems will actually be slowed down because of the added overhead, but for some problems, parallelism can provide a significant performance boost. Section 6.2 goes into the details of how to get the most from parallel execution.

Flag `--slv-parallel` *This flags enables parallel mode, allowing the synthesizer to take advantage of multiple cores. By default, the synthesizer will use one less core than the total number of cores available in your system*

Flag `--slv-p-cpus` *This flag can be used in combination to the* `--slv-parallel` *flag to indicate to the synthesizer how many cores to use.*

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 five primitive types, **`bit`**, **`int`**, **`char`**, **`double`** and **`float`**. There is a subtyping relation between three of them: **`bit`**$\sqsubseteq$**`char`**$\sqsubseteq$**`int`**, so bit variables can be used wherever a character or integer is required. **`float`** and **`double`** are completely interchangeable, but there is no subtyping relationship between them and the other types, so for example, you cannot use 1 in place of 1.0, or 0 in place of 0.0.

There are two **`bit`** constants, `0`, and `1`. Bits are also used to represent Booleans; the constants **`false`** and **`true`** are syntactic sugar for `0` and `1` respectively. In the case of characters, you can use the standard C syntax to represent character constants.

Modeling floating point Analyzing floating point values is very expensive for the synthesizer, so as of version 1.7.5, the solver provides four different representations that the analyzer can use to reason about either **`float`** or **`double`**. The flag `--fe-fpencoding` controls which representation to use.

Flag `--fe-fpencoding` *This flag controls which of three possible encodings are used for floating point values.* `AS_BIT` *encodes floating point values using a single bit; addition and subtraction are replaced with* `xor`*, and multiplication is replaced with* `and`*. Division and comparisons are not supported in this representation, nor are casts to and from integers.* `AS_FFIELD` *will encode floating points using a finite field of integers mod 7. This representation does support division, but not comparisons or casts. Finally,* `AS_FIXPOINT` *represents floats as fixed point values; this representation supports all the operations, but it is expensive. Finally, the* `TO_BACKEND` *representation uses the floating point support in the backend solver. It is the most expressive representation, and is set by default when you include the* `math.skh` *library. Its cost is very problem specific; for some problems it can be very efficient, but it can also exhibit poor scalability.*

2.2 Structs

More interesting types can be constructed from simpler types in two ways: by creating arrays of them (see Section 2.5) and by defining new types of 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;
}
```

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, but the programmer can also use named parameters to directly initialize certain fields upon allocation as shown in the following example.

Example 1. *Use of named parameters to initialize the fields of a struct.*

```
struct Point{
  int x;
  int y;
}
void main(){
  Point p1 = new Point();
  assert p1.x == 0 && p1.y == 0; //Fields initialized to default values.

  Point p2 = new Point(x=5, y=7);
  assert p2.x == 5 && p2.y == 7; //Fields initialized by 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 2. *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(); // Object C1
  Car d = c;         // after assignment d points to C1
  c.license = 123;   // the field of C1 is updated.
  assert d.license == 123;
  strange(c, d);
  assert d.license == 123; //Object C1 unaffected by call
  assert d == c;
}

void strange(Car x, Car y){
  x = new Car();  //x now points to a new object C2
  y = new Car();  //y now points to a new object C3
  x.license = 456;
  y.license = 456;
  assert x.license == y.license;
  assert x != y; //x and y point to different objects
}
```

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. All dereferences have an implicit null pointer check, so dereferencing **null** will cause an assertion failure.

2.3 Temporary Structures

There are instances where it is desirable to have the convenience of structures but without the cost of allocation and dereferencing, and without the burden of reasoning about aliasing.

The language supports *temporary structures*, which are unboxed, so they do not incur many of the usual costs associated with heap allocated structures. Temporary structures have copy semantics, so the programmer can think of them as primitive values and does not have to worry about aliasing.

One can use temporary structures as local variables and parameters by enclosing the type of the structure in vertical bars `|type|`. Temporary structures can be created with a constructor `|type|`(*args*), where *args* are named parameters just like with a normal constructor, but the keyword **new** is not used since nothing is being allocated in the heap.

Temporary structures have the following properties:

- Assignment: assignment of a temporary structure to another results in a copy.
- Equality comparison: an equality comparison of two temporary structures is equivalent to the conjunction of their field-by-field comparison.

The following example illustrates the use of unboxed functions.

Example 3.
```
struct Point{
   int x;
   int y;
}

...
|Point| p1 = |Point|(x=5, y=3); // temporary point initialized to (5,3).
Point p2 = new Point(x=3, y=2); // heap allocated point initialized to (3,2).
|Point| p3 = p1; // temporary point p3 is a copy of p1.
p3.x = 10;
Point p4 = p2; // p4 and p2 point to the same heap allocated object.
p4.x= 15;
assert p1.x == 5;
assert p2.x == 15;
assert p3.x = 10;
assert p4.x == 15;
if(??) assert p1 == p2; // equivalent to p1.x == p2.x && p1.y==p2.y
if(??) assert p1 != p2; // equivalent to !(p1==p2)
```

Interaction of temporary and heap allocated structures An assignment from a heap allocated structure to a temporary structure is interpreted as a field-by-field copy. In the above example, an assignment `p3 = p2;` would be equivalent to

```
p3.x = px.x; p3.y = p2.y;
```

Such an assignment requires that `p2` not be **null**. Similarly, an assignment from a temporary structure to a heap allocated structure is also interpreted as a field-by-field copy, with a similar assertion that the reference will not be null. Failure to satisfy the assumption will cause an assertion failure.

Similarly, an equality comparison of a heap allocated structure and a temporary structure will be equivalent to a field-by-field comparison.

Restrictions In the current version of the language, temporary structures are only allowed for local variables and function parameters. In particular, the language currently does not allow arrays of temporary structures or temporary structure fields in other structures. These restrictions are likely to be lifted in future versions of the language. Finally, structures with arrays inside them are not allowed to be temporary structures.

2.4 Final Types

Just like in Java, SKETCH has a notion of final variables and fields. Unlike Java, however, the language does not have a `final` keyword; finality is inferred based on a couple of simple rules. The rules for variables are shown below—there are analogous rules for fields of a record.

- Any variable used as an l-value cannot be final; this includes variables used as the left hand side of an assignment, variables used with pre and post increments and decrements (`++x` or `--y`), and variables passed as reference parameters to another function (see Section 2.11) .
- The iteration variable of a cannonical **`for`** loop is final within the body of the loop. A cannonical **`for`** loop is a loop of the form `for(int i=a; i<b; ++i)`, where the iteration variable is not modified inside the body of the loop.
- Arrays cannot be final.
- Global variables can only be final if they are of scalar type (not references to records).

Since assignments to final variables are disallowed by the rules, final variables must be initialized upon declaration. For fields, final fields must be initialized upon allocation through the use of named parameters to the constructor.

Expressions can also be final if they are composed from final sub-expressions. In particular:

- Final variables are final.
- A binary expression `a` *op* `b` is final if `a` and `b` are final.
- A ternary expression `a ? b : c` is final if `a`,`b` and `c` are final.
- A field dereference `e.f` is final if `e` is a final expression and `f` is a final field.

Note that expressions involving function calls or side effects cannot be final. As we will see in the next section, final types will be relevant when specifying the sizes of arrays.

2.5 Arrays

The syntax for the array 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 language will automatically check that $N \geq 0$.

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 4. *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`*.*

Dynamic Length Arrays When you declare an array of type `T[N]`, it is possible for `N` to be an arbitrary expression, as long as the expression is final as defined in Section 2.4. 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 final 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. It is important to remember, however, that any expression used as the size of the array must be final, so in particular, they cannot involve any function calls.

When the size of the array needs to be computed by the function itself, there are two ways to proceed. One option is to give an over approximation of the size of the array as indicated by the example below. The other option is to package the array into a **struct** as shown in the next paragraph.

Example 5. *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. The function uses a reference parameter (see Section 2.11) to return* `outsz`*. We may use the function as follows:*

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

The function uses the bulk array access `tmp[0::sz]`*, which will be defined properly later in the section. A cleaner way of writing this example is to use records as shown in the next paragraph.*

Array fields Records can also have arrays as fields. The expression for the array size can involve any final expression in scope, which in practice means final expressions involving global variables and other final fields. Keep in mind that if a field is final, then it must be initialized by the constructor of the record.

Example 6. *Using array fileds, we can write a cleaner version of the filter function from before:*

```
struct Array{
        int sz;
        int[sz] A;
}


Array filter(Array arr){
        int outsz = 0;
        int[arr.sz] out;
        for(int i=0; i<arr.sz; ++i){
                if(arr.A[i]%2 == 0){
                        out[outsz++] = arr.A[i];
                }
        }
        return new Array(sz=outsz, A=out[0::outsz]);
}
```

One interesting point to note is the size of `out`*; because it uses* `arr.sz`*, that forces both the variable* `arr` *and the field* `sz` *to be final. The field* `sz` *was already final because it was used in the size of field* `A`*, but now that* `arr` *is also required to be final; assigning anything to it inside the function would be illegal and would be flagged by the type checker. Finally, the function uses the bulk array access* `out[0::outsz]`*, which will be defined in the next section.*

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* `bnd-arr-size`^{2}.

Multi-dimensional arrays In sketch you can declare a multi-dimensional array as follows.

`T[`sz_0`, `sz_1`, ..., `sz_n`] A;`

You can then access an element `A[`i_0`,`i_1`,...,`i_n`]`, where $0 \le i_j < sz_n$.

Multi-dimensional arrays are actually just syntactic sugar for nested arrays; so a declaration like the one above would be desugared to the one below.

`T[`sz_n`]...[`sz_1`][`sz_0`] A;`

And similarly, the access `A[`i_0`,`i_1`,...,`i_n`]` is desugared to `A[`i_0`][`i_1`]...[`i_n`]`. The nested array notation and the multi-dimensional array notation are completely interchangeable, so for example, it is fine to pass a nested array to a function expecting a multi-dimensional array, and vice-versa.

The main advantage of the multi-dimensional array notation over the nested array notation is that the bounds in the declaration appear in the same order as the indexes in the array access, instead of being reversed as happens with the nested array notation.

Bulk array access The indexing operation we just saw will read a single element from an array. The SKETCH language also includes 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 greater than or equal to zero and `x + M`≤`N`, then the expression `a[x::M]` produces an array of type `T[M]` containing the elements `a[x], ..., a[x+M-1]`.

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`≥`N`. If `M`≠`N`, the rhs will be padded with zeros or nulls according to the rules in Section 2.6.

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];
```

String literals (Character array constants) Like C, sketch supports strings in double quotes as a shorthand for a null-terminated character arrays. So for example, the string `"a string"` is a shorthand for the array constant:

```
{'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0'}
```

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 2.6. 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 == b when a and b have type T[n] ⇒ n==m ∧∀ i < n a[i]==b[i]
a == b when a is of type T[n] and b is of type T ⇒ n==1 ∧ a[0] == b
```

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

Example 7. *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 2.6.

2.6 Automatic Padding and Typecasting

When arrays are assigned or passed as parameters, they can be implicitly typecast through padding. Padding is also supported by bit vector operations, but not for reference parameters. 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 8. *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}`*.*

Example 9. *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}}`

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 or zero depending on the context.

Example 10. *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}`*.*

There are some cases where it is desirable to assign a constant to an array of size `n`, where `n` may be an input to the function. If `n` is zero, however, the constant will automatically be cast to an empty array of size zero in order to avoid an error.

Example 11. `int[n] x;`

```
...
x = 0; // This will zero out the array; will not cause an error when n==0
```

The casting of scalars to arrays of size 1 also works in the context of multidimensional arrays.

Example 12. *The following block of code illustrates padding for multi-dimensional arrays.*

```
struct Car{ ... }
...
Car[2] x = {new Car(), new Car()};
Car[4][3] y = x;
```

```
assert y[0][0] == x[0];
assert y[1][0] == x[1];
assert y[1][1::3] == {null, null, null};
assert y[2] == {null, null, null, null};
```

To understand the example above, it is worth thinking of an array assignment of the form `y=x` where `y` is of type `T[N]` as a loop of the form

```
for(int i=0; i<N; ++i){
   if(i<M)//where M<=N is the size of x
     y[i] = x[i];
   else
     y[i] = pad(T);
}
```

So in the example above, `y[i]` will be of type `Car[4]` and `x[i]` will be of type `Car`, which means each assignment involves an implicit typecast from `Car` to `Car[4]`.

2.7 Explicit Typecasting

The SKETCH language also offers some limited explicit typecasting. In particular, the language offers only four 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.
- If using `--fe-fpencoding AS_FIXPOINT`, explicit casts from integers to floating point values and vice versa are also allowed.
- An object of an algebraic data type can be explicitly typecast to one of its variants. The system will automatically introduce an assertion to ensure that the value is of the right type. For example, in the `Tree` example from Section 2.8, one can cast a value of type `Tree` into a value of type `Leaf`, but if the value is a `Branch` instead of a `Leaf`, the cast will trigger an assertion failure.

Example 13. *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 14. *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.*

2.8 Algebraic Data types

Sketch also supports algebraic data types, or case classes as they are known in Scala. To illustrate the use of algebraic data types (ADTs), consider the `Tree` ADT defined below.

```
adt Tree{
    Leaf{ int value; }
    Branch{ Tree left; Tree right; }
}
```

`Tree` is an ADT by virtue of having two variants that extend it: `Leaf` and `Branch`. Both `Leaf` and `Branch` can be instantiated just like any other kind of record, and they are both subtypes of `Tree`. For example, the code below instantiates a small tree.

```
Tree t1 = new Leaf(value=5);
Tree t2 = new Leaf(value=6);
Tree t3 = new Leaf(value=7);
Tree t = new Branch(left=t1, right=new Branch(left=t2, right=t3));
```

The fact that Leaf and Branch extend `Tree` causes `Tree` to be abstract, so `Tree` itself cannot be instantiated.

Pattern matching in SKETCH is done through the **`switch`** construct. The basic syntax is as follows:

```
switch(var){
   case C1:
   ...
   case Cn:
}
```

The argument `var` passed to switch must be a variable whose type is an ADT, and each case must be labeled with a different variant `Ci`. Within that case, the variable `var` will automatically acquire the type corresponding to the case. This is illustrated by the function below:

```
int sum(Tree t){
   int v=0;
   switch(t){ // There is an implicit assertion that t is not null.
      case Leaf:{
        /*t is now of type Leaf in this block */
        v = t.value;
      }
      case Branch:{
        /*t is now of type Branch is this block */
        v = sum(t.left)+sum(t.right);
      }
   }
   /*Outside the switch, t goes back to being of type Tree*/
   return v;
}
```

The **`switch`** construct looks a lot like the **`switch`** construct in C or Java, but its semantics are very different. First, the cases correspond to the possible types of `t`, as opposed to different values of an integer expression. Second, the argument to switch must be a variable, and the type of this variable will be different in each of the cases. And finally, the cases do not “fall through” the way they do in C; only one case can match, and only the block of code associated with the matching case will execute. Note that there is an implicit assertion that t is not null.

The language allows hierarchies of ADTs. For example:

```
adt ASTNode{
    adt Expression{
        Plus {Expression left; Expression right;}
        Times {Expression left; Expression right;}
        Num{ int n; }
        Var{ int id; }
    }
    adt Statement{
        Assign{ int lhs; Expression rhs; }
        IfStmt{ Expression cond; Statement tpart; Statement epart; }
        WhileStmt{Expression cond; Statement body;}
    }
}
```

So in this case, one can create instances of `Plus`, `Times`, `IfStmt` and `WhileStmt`, but not of `ASTNode` or of `Expression` and `Statement`. When pattern matching, however , one can have cases for `Expression` or `Statement` and those cases will match on any variant that is a subtype of them.

The following are some additional rules associated with ADTs.

- All fields of an ADT are immutable.
- Records representing an ADT can have fields, and those fields will be inherited by all the variant types.
- A parent and a variant cannot both have a field with the same name.
- The type extended by a variant class should exist and be defined in the same package, so ADT definitions cannot span multiple packages (see Section 2.17 for more on packages).
- The expression that is passed to switch(e) should always be a variable; the labels for the cases should be variants of the type of that variable, and the variable cannot be null.
- Switch statements should cover all the cases.
- Implicit casting occurs from a subtype to a super type.
- Explicit casts can be used only from a supertype to a subtype. An incorrect cast will cause an assertion failure.

Equality Although ADT values look like references, and for the most part behave like references, the one exception is equality (and not-equals) comparison. Two ADT values will be considered to be equal if all their fields are equal. For example, if you have two definitions as in the code below:

```
Tree t1 = new Leaf(value=5);
Tree t2 = new Leaf(value=5);
```

The variables `t1` and `t2` are considered to have the same value, and cannot be distinguished from each other. So the following assertion will pass:

```
assert t1 == t2;
```

So will the assertion

```
Tree b1 = new Branch(left = t1, right= t2);
Tree b2 = new Branch(left = t1, right= t2);
assert b1 == b2;
```

For recursive ADTs, Sketch will automatically generate a recursive comparison function to test equality, and will be subject to the inlining bounds given through the command-line flag `bnd-inline-amnt`. In the context of ADTs, the constant **`null`** is just a special value that can only be compared against. ADT variables are all initialized to **`null`** if they are not explicitly initialized, so you can use comparisons such as

```
if( t1 == null){

}
```

to check if a variable has been initialized, but any other operation that you try to do on a **`null`** value (reading a field, explicit casting or switching on it) will lead to an assertion failure.

2.9 Struct inheritance

In some cases, we want to make use of the ability of ADTs to define sub-types and to use switch to distinguish between cases, but we do not want to be restricted by the immutability requirement. In this case, we can use **`struct`** inheritance to achieve that goal. For example, we can write:

```
struct Tree{
        int x;
}

struct Leaf extends Tree{
    int value;
}
struct Branch extends Tree{
    Tree left;
    Tree right;
}
```

We can use the above Branch, Leaf and Tree types the same way we would have if we had defined them using the `adt` syntax. The difference is that now, the `Leaf`s and `Branch`es will be mutable, so you can change their value after they have been created. There is a cost to this, though, in that the synthesizer will be able to handle immutable ADT values much more efficiently than these mutable structures. Also keep in mind because these are **`struct`**s, equality behaves just like with other structs.

2.10 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 support a C-like **`switch`** statement, since **`switch`** is used for pattern matching instead (see Section 2.8). The language also does not support `continue` and **`break`**, although they can easily be emulated with **`return`** by using closures (see Section 2.13).

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 15. *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 (as long as there are no return statements in its body). For the third loop, the unroll factor will be controlled by the flag, but will never exceed seven.*

2.11 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.*

Flag `--bnd-bound-mode` *The solver supports two bound modes:* `CALLSITE` *and* `CALLNAME`*. In* `CALLNAME` *mode (the default), the flag* `bnd-inline-amnt` *will bound the number of times any function appears in the stack. In the* `CALLSITE` *mode, the* `bnd-inline-amnt` *flag will bound the number of times a given* call site *appears on the stack, so if the same function is called recursively multiple times, each site is counted independently.*

Reference Vs. Value Parameter Passing By default, parameter passing is done by value, so even if parameters are modified inside the function, the change will not be visible outside. We can make changes to a parameter visible to the caller by prefixing the formal parameter with the keyword `ref` as shown in the example below.

Example 16. *Prefixing the formal parameter with* `ref` *makes changes visible outside the function.*

```
void foo(int in, ref int out){
        in = in + 1; // changes to in are not visible to the caller
      out = in + 1; //changes to out are
}
harness void main(int x){
      int y = x;
        int z = y+10;
      foo(y, z); // call to foo can change z but not y
      assert y == x && z == x+2;
}
```

A very important point about `ref` parameters is that their semantics is not actually pass-by-reference; it is copy-in-copy-out. If you pass only local variables by reference and you never alias `ref` parameters, copy-in-copy-out semantics and pass-by-reference semantics are indistinguishable, and the compiler will actually generate pass-by-reference code. However, if you alias parameters or if you pass a field of a record by reference, the copy-in-copy-out semantics will become apparent.

Example 17. *This example illustrates copy-in-copy-out semantics.*

```
void foo(ref int p1, ref int p2){
      int t=p1;
      p1 = p1 + 1;
      p2 = p2 + 2;
      assert p1 == t + 1;
}
harness void main(int x){
        int z = x;
      foo(z, z);
      assert z == x+2;
}
```

Note that copy-in-copy-out semantics make it possible to reason about `foo` *in isolation; the assertion will hold regardless of whether or not the user passes the same parameter to* `foo` *twice. The copy-out happens in the order of the parameter list, so the final value of* `z` *is the value of* `p2`*.*

Another very important point is that the `ref` modifier can also be used in generator functions ((see Section 4), but in that case, `ref` parameters do have pass-by-reference semantics.

Implicit size parameters When passing arrays as parameters, it is common to have to separately pass the size of the array, so a typical function signature will look like this:

```
double[n] foo(int n, double[n] in)
```

SKETCH allows you to declare the size parameter `n` as an *implicit parameter* so that you do not have to pass it when calling the function.

```
double[n/2] foo([int n], double[n] in){//brackets signify implicit parameter
     return in[0::n/2];
}
harness void main(int n, double[n] in){
   double[n/2] res = foo(in); //we don't have to pass n.
   double[n] res = foo(2*n, in);//but we can if we want.
}
```

The syntax for a function definition that uses implicit parameters is shown below.

ret_type name([*implicit args*], *args*){
  body
}

Specifically, implicit parameters must be at the beginning of the parameter list, and the complete list of implicit parameters must be enclosed in square brackets.

There are two important semantic requirements for implicit parameters. First, all implicit parameters must be of integer type, since they must correspond to sizes of arrays passed as parameters. The second requirement is that every implicit parameter must be equal to the size of at least one array argument.

Example 18. *The following definitions correspond to legal uses of implicit parameters:*

```
int[n] foo([int n, int m], int[n] a, int[m] b)...
int[m] foo([int n], int[n] a, int m, int[m] b)...
int[n+m] foo([int n, int m], int[n][m] a, int[m] b)...
int[n+m] foo([int n, int m], int[n][m] a, int[2*m] b)...
```

By contrast, the following are all illegal:

```
int[n] foo([int n, int m], int[n] a)... // m is not used
int[n] foo([int n], int[n*2] a)... //n is not the size of any array param
int[n+m] foo([int n, int m], int[n][m*2] a, int[m-1] b)... //m is not the size of any array param
```

When calling a function with implicit parameters, there are two rules to keep in mind.

- You can decide to pass all implicit parameters, or you can pass none of them, but you cannot selectively pass only a subset of them.
- If a given implicit parameter is used by multiple array parameters, all array parameters must be consistent regarding the value of that parameter.

Example 19. *Given the following function definition and declarations*

```
void foo([int n, int m], int[n] a, int[m][n] b)...
...
int[5] w;
int[5] x;
int[3][5] y;
int[10] z;
```

The following calls are valid:

```
foo(x, y); // n==5, m==3
foo(w, x); // n==5, m==1
foo(10, 3, z, y); // Involves an implicit cast for y.
```

By contrast, the following are all illegal:

```
foo(z, y); //different value of n for the two arrays
foo(3, z, y); //Passes only a subset of implicit parameters
```

2.12 Function parameters

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

Example 20. *Functions as 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.

One important point to notice about function parameters is that the signature does not specify what parameters it expects. This gives the language some flexibility, and in some cases allows one to make up for the fact that we don't have generics. However, it also has an important implication. Namely, when a function parameter is called, it is not possible to know which parameters will be reference parameters and which parameters will not, so any variable that is passed to a function that came as a parameter will be considered non-final.

2.13 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. One important aspect of closures in Sketch is that because functions cannot be returned by other functions or written into the heap, a local function can never escape the context in which it was declared. This allows local functions to modify local variables defined in their host function without any messy semantic issues.

Finally, local functions can only be used after they are declared, unlike top level functions for which the order of declaration does not matter. One important implication of this is that you cannot define mutually recursive local functions.

```
harness void main(int N, int[N] A){
      int foo(int i){
        if(i>0){
             return moo(i-1); // not allowed because moo has not been declared.
        }
        return i;
      }
      int moo(int i){
         return foo(i);
      }
}
int fooTop(int i){
      if(i>0){
        return mooTop(i-1); // This is ok, for top level functions the
      }                     // order of declaration does not matter.
      return i;
}
int mooTop(int i){
     return fooTop(i);
}
```

2.14 Lambda Functions

Lambda functions are functions without a name. They are also known as *anonymous functions*. They allow the creation of a function without specifying a name and return type. The syntax to define a lambda is `(vars) -> expression`. Lambdas are used as actual parameters of high-order functions or initializers to variables of type `fun`.

Example 21. *Passing a lambda to a high order function.*

```
int apply(fun f, int x) {
    return f(x, 5);
}

harness void main() {
    int result = apply((x, y) -> x + y, 7);

    assert result == 12;
}
```

Example 22. *Defining a variable of type* `fun`.

```
harness void main() {
    int t = 3;
    fun foo = (x) -> t * x;

    assert foo(2) == 6;
}
```

Lambda functions must be defined and used in the local scope or in local functions. The body of the lambda function can be any valid expression in the language. The formal parameters of the lambda function cannot be used as l-values.

Lambda functions cannot be defined inside **struct** nor can they be returned from a function. These are the same restrictions on the usage of local function (see Section 2.13). Variables of type `fun` are final and cannot be modified once they are initialized.

Example 23. *Using two local variable constructs of different types inside a local function.*

```
harness void main() {
    bit t = 1;
    int a = 2;
    int b = 3;

    int foo() {
        fun g = () -> $(bit) ? 5 : $(int);
        return g();
    }
    assert foo() == 3;
}
```

Example 24. *Using a local variable construct inside a local function that is passed to a high-order function.*

```
int apply(fun f) {
    return f();
}

harness void main(int x) {
  int a = 1;
  int b = 2;

  int foo() {
      fun f = () -> $(int);
      return f();
  }

  int t = apply(foo) * ??;

  assert t == x + x;
}
```

2.15 Polymorphism

Starting with version 1.7.1, Sketch has full support for polymorphic functions and generator functions (Section 4). A polymorphic function can be declared by including type parameters in its signature:

type name<*typeParams*>(*type pname*, ...){ ... }

For example, the following is a well known forall function applied to arrays:

```
void forall<T>([int n], fun f, ref T[n] x){
    for(int i=0; i<n; ++i){
        f(x[i]);
    }
}
```

When calling `forall`, `T` can be used in place of any data-type, including primitive types, structures, adts, and arrays thereof. The type `T`, however, cannot be instantiated to a function type.

In order to use a polymorphic function, you just need to call it as you would any other function.

Example 25. *Use of a polymorphic function.*

```
harness void main(){
    int sum=0;
    generator void add(int t){
        sum += t;
    }
    forall(add, {1,2,3,4,5});
    assert sum == 15;
    sum = 0;
    forall( (u) -> forall(add, u),  { {1,2,0} , {3,4,5} } );
    assert sum == 15;
}
```

Starting in Sketch 1.7.5, the language also supports polymorphic structures and algebraic data types (ADTs). You can declare a polymorphic struct by adding <*typeParams*> after the name of the struct.

For example, the Array library (Section 7.4) defines an `SArray` structure as:

```
struct SArray<T>{
    int n;
    T[n] val;
}
```

The structure can be used in the context of a program with different types for the type variable `T`. For example, in the code below, the type is instantiated as `int` for `ar1`, and as `SArray<int>` in the case of `ar2`. Note that in the example we also define a polymorphic function `len`, that works for `SArray<T>` for any valid `T`.

Example 26. *Use of a polymorphic structure.*

```
include "array.skh";

int len<T>(SArray<T> ar){ return ar.n; }

harness void main(){
    SArray<int> ar1 = new SArray<int>(n=5, val={1,2,3,4,5});
    SArray<SArray<int>> ar2 = new SArray<SArray<int>>(n=1, val={ar1});
    assert ar1.val[0] == 1;
    assert ar2.val[0] == ar1;
    assert len(ar1) == 5;
}
```

The approach also works for ADTs; for example, consider the List ADT in the `list` package (Section 7.5).

```
adt List<T>{
    Cons{ T val; List<T> next; }
    Nil{}
}
```

Note that the type parameters are only declared at the top level for the overall `List<T>` type. When invoking one of the constructors, we must provide a type parameter for the constructor. For example, the following function is part of the list library:

```
List<T> add<T>(List<T> lst, T val){
    return new Cons<T>(val=val, next=lst);
}
```

The function adds an element to the beginning of the list by constructing a new `Cons<T>` value.

Limitations The current implementation of polymorphism has the following limitations.

1. For structures and algebraic data types, the type parameter is currently not allowed to be an array. We expect this limitation to be lifted in future releases of the language.

2. Constructors for structures and ADTs do require the type parameters to be passed explicitly. We expect this limitation to be lifted in future releases of the language.

3. In the case of functions, the compiler relies on a very local inference algorithm in order to discover the type parameters. This means that the function must be used in a context that provides enough information about the expected types. This is illustrated by the example below.

```
//Consider the two functions below.
T read<T>(List<T> in){ ... }
List<T> gen<T>(){ ... }
void foo(List<int> lst){
    int x = read(lst); // From the context it is clear that T=int;
    int y = read(gen()); //In this case too, the context is enough
                         //to determine that T = int;
    read(gen()); // However, this would lead to a type error
                 // because the type of T is ambiguous.
}
```

4. In some cases, the type inference algorithm may infer types that are more specific than what you need because the type inference algorithm is not very aggressive in applying typecasts. For example, while most integer constants have type **int**, the constants 0 and 1 have type **bit**, so if you have a function such as `List<T> single<T>(T in);` and you call `single(0)`, the compiler will infer that T is **bit**, so if you actually wanted a List<int>, then you should either cast 0 to (**int**)0 or assign it to a variable of type **int**.

2.16 Uninterpreted Functions

SKETCH also supports uninterpreted functions, which can be defined with the following syntax.

ret_type name(*args*);

An uninterpreted function is a function whose body is unknown, so from the point of view of the synthesis and verification engine; there is nothing known about this function other than the fact that it is a pure function, so when fed with the same inputs it will produce the same outputs. Importantly, if you are using an uninterpreted function to model some complex routine in the program, you need to make sure that routine behaves as a mathematical function; i.e. it should not access the heap or global variables. For this reason, we restrict uninterpreted functions so they can not involve structures.

2.17 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 27. *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
```

2.18 Global variables

The sketch language supports global variables with a few important restrictions.

- First, global variables are always private to the package in which they are defined; they cannot be made public, although you can have functions in a package that read and write to a given global variable.
- The second restriction is a consequence of the fact that only scalar global variables can be final; this means that global arrays must have constant dimensions since, in that scope, constants are the only thing that can be final.
- The initializers for global variables can only involve side-effect-free expressions of constants and final global variables. An expression that allocates a struct (*e.g.* `new C(a=x)`) is considered a side-effect-free expression as long as the arguments to the constructor (*e.g.* `x`) are side effect free. Division, array access and function calls are not considered side-effect free since they may fail.

2.19 Annotation System

The SKETCH language includes an annotation system that is meant to simplify the process of adding language extensions. The general syntax for annotations is as follows:

`@Name(parameter-string)`

`Name` is the name of the annotation and `parameter-string` is a string describing the parameters of the annotation. Annotations are currently only supported for function and record definitions.

Annotations are preserved throughout the compiler chain all the way to code generation. They can be particularly useful if you are writing your own custom code generator (see Section 6.4). In that case, you can just introduce annotations in your code, and your code generator can query the `Function` and `StructDef` AST nodes to get their annotations (see the sample code generator provided for how to do this). There are, however, a handful of annotations that have special meaning to the compiler. Theey are listed below.

- `@Native(`*code*`)` overrides code generation for the annotated function (see Section 2.20).
- `@NeedsInclude(`*inc*`)` tells the code generator that a particular function requires some specific header file to be included (see Section 2.20).
- `@Immutable()` declares a given structure as immutable.
- `@DontAnalyze()` tells the synthesizer that a given harness should not be analyzed. This is useful if you have a harness for which you want to generate code, but which you know to be too big and complex to analyze.
- `@Private()` indicates that a function should not be used outside the package where it is defined.
- `@Gen()` This is used to annotate a special class of functions that have a special meaning for the analysis engine. You should not be using this annotation in your own code.

2.20 The `@Native` annotation

Two important annotations are `@Native` and `@NeedsInclude`. The first is `@Native`, which allows the user to override the standard code generator and tell the synthesizer exactly what code to synthesize for a particular function. The second one is `@NeedsInclude` which is used to tell the code generator that a particular function requires some specific header file to be included.

For example, the following code shows how the two `@Native` annotations can be used to write a set of routines that read form a file.

Example 28. *In* SKETCH*, one can use the following classes to model the process of reading from a file.*

```
int NDCNT=0;

int getND_private(int i);
int getND(){
    //Every time this function is called
    //it produces a new non-deterministic value.
    return getND_private(NDCNT++);
}

struct FileHandle{
    int maxReads; //Number of values left in the file.
}

FileHandle getFile(){
    //Number of values in the file is some non-deterministic value.
    return new FileHandle(maxReads=getND());
}

bit moreValues(FileHandle fh){
    //maxReads should never drop below zero.
    assert fh.maxReads >= 0;
    return fh.maxReads!=0;
}

int readInt(FileHandle fh){
    //Reads past the end of the file are not allowed.
    assert fh.maxReads > 0;
    --fh.maxReads;
    return getND();
}
```

The `FileHandle` *is initialized with the maximum number of values to read. Every time the client calls* `readInt`*, the synthesizer reads checks if the maximum number of reads has been reached and reads another non-deterministic value. This definition of the operations on a file is very good if we are interested in synthesizing or verifying a client that needs to read a file and do something with its contents. However, if we want to generate code to read real files, the class above is not so useful.*

Using the `@Native` *annotations, however, we can instruct the synthesizer on how to generate code for the structure and functions above. For example, the code for the* **`struct`** *would be as follows:*

```
struct FileHandle{
    int maxReads;
    @NeedsInclude("#include <fstream>")
    @NeedsInclude("#include <string>")
    @Native("ifstream in;")
    @Native("int last;")
    @Native("bool goon;")
    @Native("FileHandle(const string& s):in(s.c_str()){ in>>last; goon = !in.eof() && !in.fail(); }")
    @Native("int readInt(){ int x = last; in>>last; goon = !in.eof() && !in.fail(); return x;}")
}
```

The functions annotations are used to introduce additional fields and methods which are invisible to the analysis engine, but which are needed by the generated code.

With their annotations, the `moreValues` *and* `readInt` *functions are as follows:*

```
@Native("{ _out = fh->goon; }")
bit moreValues(FileHandle fh){
    assert fh.maxReads >= 0;
    return fh.maxReads!=0;
}


@Native("{ _out = fh->readInt(); }")
int readInt(FileHandle fh){
    assert fh.maxReads > 0;
    --fh.maxReads;
    return getND();
}
```

When analyzing code, the annotations are invisible to the synthesizer, and it will focus on the high-level model in the body. When generating code, on the other hand, the code generator will produce the code instructed by the `@Native` *annotation.*

The `@Native` annotation allows the programmer to use simple models in place of very complex or low-level functions. It is the responsibility of the programmer to ensure that the model matches the relevant behavior of the code that is being generated.

More generally, if you want to write custom extensions to the SKETCH synthesizer, you can use annotations to pass information to your custom extension without affecting any of the existing synthesizer infrastructure.

2.21 Casting of Expressions

Sketch provides casting of expressions in high-order function calls. These expressions are cast as lambda functions. The expressions that can be automatically cast are the ones that do not use formal parameters inside the function. This means that the actual parameters passed to the cast function are ignored when the body of the lambda gets executed.

This feature is very useful in High-Order Generators (see Section 4.4).

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

If the users wants to pass some variables to the high-order generator, they can use the Local Variable construct (see Section 4.3). This expression will be cast as a lambda expression before being passed to the high-order generator.

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

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 Harnesses and function equivalence requirement

A program in sketch can have multiple harness functions each encoding different requirements of the problem. Sketch will guarantee that all harnesses will run to completion without triggering assertion failures for all inputs within bounds. Harness functions are not allowed to take heap allocated objects as inputs and all global variables are reset to their initial values before the evaluation of each harness. Because of this, the order of evaluation of the harnesses does not matter.

Sketch also allows the programmer to express that a function—we call it the implementation—must be functionally equivalent to another function—the specification—by writing **implements***fname* at the end of the signature of the implementation function. When the specification and implementation do not access global variables, the equivalence constraint created by implements can be seen as syntactic sugar for a harness that calls both the implementation and the specification and compares their results. However, using **implements** is preferable to writing such a harness because it lets the compiler know that it can replace one function for the other when reasoning about the program. The implements directive imposes stronger constraints than harness; same function cannot have a harness and an **implements** directive.

If the implementation or the specification access global variables, then their equivalence must hold for all possible values of those global variables—not just for their initial values—and they must leave the global variables in the same state after they terminate. This is to ensure that replacing the specification with its implementation will always be semantics preserving. For this same reason, the spec and the sketch cannot access global variables that point to heap allocated values.

Example 29. *As a simple example of the use of harnesses and* **implements***, consider the following sketch.*

```
int count=0;
int twox(int x, int y){
        ++count;
        return x + x;
}

int expr1(int x, int y){
        return x*?? + y*??;
}
int expr2(int x, int y) implements twox{
        count = count+??;
        return x*?? + y*??;
}

harness void foo(){
        assert expr1(5,2)*expr2(2,4)== 24;
}
```

The harness imposes a constraint on the product of the two functions. The **implements** *in the declaration of* expr2 *imposes the additional constraint that* expr2 *must be equivalent to* twox*. Because* twox *modifies the global variable* count*,* expr2 *must modify the variable in the same way.*

Implementation note As of the current SKETCH release, there is a requirement that the specification should not have any unknowns. It is likely that this requirement will be relaxed in future releases. Also, when two functions are related by an **implements** directive, the compiler assumes that the specification is simpler than the implementation. The compiler exploits this by replacing every call to the implementation with a call to the specification during the analysis phase. This generally leads to big performance improvements for the solver, but it means that if the implementation calls itself recursively the system will not catch bugs that lead to infinite recursion.

3.2 Assumptions

Starting in SKETCH 1.6.7, the compiler supports standard `assume` statements. The semantics of `assume`*cond*; dictate that when the condition *cond* is false, execution stops and any subsequent assertions are ignored. When a function is called by another function, any assumptions in the callee become assertions from the point of view of the caller; *i.e.* the caller must guarantee that the inputs and environment it passes to the callee satisfy the assumptions. *Note that this is not true of generator functions (Section 4)*; as we will describe later, generators get inlined into their calling context, so any `assume` statement in the generator is an assume statement in the function that invokes the generator.

Also, when a function implements a specification, its assumptions must be weaker than those of the specification. In other words, any input that is legal for the specification must be legal for the implementation.

Example 30. *Assume and implements*

```
harness int foo(int x){
        assume x > 10;
        int t= x-10;
        assert t > 0;
        return t;
}

int moo(int x) implements foo{
        assume x > 3;
        int t = x-??;
        assert t > 0;
        return t;
}


harness void main(int x){
        assume x > 5;
        int t = ??;
        moo(x+t);
        minimize(t);
}
```

In the example above, the harness `foo` *assumes* `x>10`*, which allows it to satisfy the assertion* `t>0`*. In the case of* `moo`*, the unknown will resolve to* `10` *because of the constraint of equivalence with* `foo`*. Note that the assumption in* `moo` *is not strong enough to prove the assertion, but because* `moo` *implements* `foo`*, it inherits its preconditions.*

3.3 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.

3.4 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.*

3.5 Minimizing Hole Values

In many cases, it is useful to ask the synthesizer for the smallest constant that will satisfy a specification. For such cases, the synthesizer supports a function `minimize(`e`)`, which asks the synthesizer to make e as small as possible. More specifically, the synthesizer will find a minimal bnd such that $e < bnd$ for all inputs. If the program includes multiple `minimize` statements, the synthesizer will find a locally minimal set of bounds such that there is no other set of bounds that is strictly better than the one found.

Flag `--bnd-mbits` *The flag* `bnd-mbits` *tells the synthesizer how many bits to use to represent all bounds introduced by* `minimize(e)` *(default 5). Note that the largest value of* `(e)` *will be less than the bound, so if* `e` *can have value n, the bound needs enough bits to be able to reach $n+1$.*

Example 31. *For example, consider the following simple program:*

```
harness void main(int i, int j){
        int i_orig=i, j_orig=j;
        if(i > ??){ // we'll call this unknown u1
                i = ??; // u2
        }
        if(j > ??){ // u3
                j = ??; // u4
        }
        if(i_orig > 3 && j_orig > 3)
           assert 2*i + j > 6;
        minimize(i);
        minimize(j);
}
```

In the program above, synthesizer will try to minimize the upper bound on `i` *(we'll call it b_i) and the upper bound on j (b_j). One possible solution is to have $u1 = 3$, $u2 = 4$, $u3 = 0$, $u4 = 0$. This will allow the upper bounds (b_1, b_2) to be $(5, 1)$. A different possible solution is to have $u1 = 0$, $u2 = 0$, $u3 = 0$, $u4 = 7$, which will allow the upper bounds to be $(1, 8)$. While the first set of bounds appears better than the second, they are actually incomparable. By contrast, a solution that had $(b_1, b_2) = (6, 1)$ would not be allowed because the solution with bounds $(5, 1)$ is strictly better.*

The synthesizer has some syntactic sugar on top of `minimize` to support the synthesis of a minimal number of statements, a common idiom in SKETCH. For example, the code below will produce the minimal number of assignments required to swap `x` and `y`.

```
void swap(ref bit[W] x, ref bit[W] y){
    minrepeat{
        if(??){ x = x ^ y;}else{ y = x ^ y; }
    }
}

harness void main(bit[W] x, bit[W] y){
    bit[W] tx = x; bit[W] ty = y;
    swap(x, y);
    assert x==ty && y == tx;
}
```

4 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));
}
```

4.1 Recursive Generator Functions

Generators derive much of their expressive power from their ability to recursively define a space of expressions. For example, the code below shows how to use a recursive generator to define a context free grammar of possible expressions.

Example 32. *Recursive Generator*

```
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);
}
```

One must be careful when defining recursive generators, however, because the way the generator is defined can have a dramatic impact on the solution time of the resulting code. In particular, there are two aspects that the writer must keep in mind when writing a generator: recursion and symmetries.

Recursion control in generators The compiler handles recursive generators by inlining them number of times as guided by the `bnd-inline-amnt` flag. This simple approach can cause problems if recursive generators are not written carefully. For example, an alternative way of writing the generator above is shown below.

Example 33. *Inefficient recursive generator*

```
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;}
   if(t == 3){return rec(x,y,z) * rec(x,y,z);}
   if(t == 4){return rec(x,y,z) + rec(x,y,z);}
   if(t == 5){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);
}
```

Both generators describe the same grammar, and therefore in principle the same space of possible expressions, but the second generator will cause problems because each recursive call to `rec` will be inlined independently, so each level of inlining will increase the size of the program by a factor of six, instead of only a factor of two.

Another potential issue with recursive generators is that the amount of inlining is controlled by the same flag used to control inlining of functions during analysis. This can be problematic because recursive functions in the program will often require much more inlining than generators. To address this problem, the user can take additional control over inlining by explicitly adding a bound parameter into the generator as shown below.

Example 34. *Generator with manual inlining control*

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

   int a = rec(x,y,z, bnd-1);
   int b = rec(x,y,z, bnd-1);
   ...
}
```

The synthesizer performs partial evaluation in tandem with inlining, so if we call rec with a constant value for the `bnd` parameter, the synthesizer will stop inlining when it determines that this parameter will be less than zero.

Avoiding symmetries Another aspect to be careful with when defining recursive generators are symmetries. These happen when different assignments to unknown values can result in the exact same expression. An important source of symmetries are commutative and associative operations. For example, consider two generators shown below.

Example 35. *Effect of symmetries on generators*

```
generator int sum(int x, int y, int z, int bnd){
    assert bnd > 0;
   generator int factor(){
       return {| x | y | z|} * {| x | y | z | ?? |};
   }
   if(??){ return factor(); }
   else{return sum(x,y,z, bnd-1) + sum(x,y,z, bnd-1);}
}

generator int sumB(int x, int y, int z, int bnd){
    assert bnd > 0;
    generator int factor(){
        return {| x | y | z|} * {| x | y | z | ?? |};
    }
    if(??){ return factor(); }
    else{ return factor() + sumB(x,y,z, bnd-1);}
 }
```

Both represent the same space of expressions, but the generator `sumB` forces a right-associativity on the expression, whereas the generator `sum` can produce all possible associations, making the generator `sumB` more efficient than `sum`. Additionally, in `sumB` the `bnd` parameter has a clear meaning: it is the number of terms in the sum, whereas in generator `sum`, the parameter `bnd` is the depth of the AST, which is not as straightforward to map to something meaningful to the programmer.

4.2 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)? |};
}
```

4.3 Local Variables Construct

Sketch supports the use of the $(type) construct to instruct the synthesizer to consider all variables of the specified type within scope when searching for a solution.

```
harness void main(int x) {
    int a = 2;
    double b = 2.3;

    assert x * $(int) == x + x; // $(int) === {| 0 | a | x |}
}
```

The value of *type* can be any of the primitive types (see Section 2.1) or any user defined type. The default value of any primitive type will also be considered as one of the choices. Local variables inside a function and its formal parameters are considered within scope of the construct. If the construct is used inside a local function, the local variables and formal parameters of the functions where it is defined are also within scope of the construct.

4.4 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);
    }
}
```

4.5 repeat construct

The pattern illustrated by the `rep` generator above—where the user wants to generate multiple statements, all of which can be generated from the same code with holes—is quite common. Sketch includes a repeat construct to do exactly this:

```
repeat(N){
    stmt
}
```

This construct is just syntactic sugar for the recursive *rep* generator shown earlier. A big difference between using **repeat** and writing your own `rep` generator is that with repeat, the maximum number of repetitions will be dictated by the loop unrolling bound `bnd-unroll-amnt`, whereas if you write your own recursive generator, the maximum number of repetitions will be dictated by the inlining bound `bnd-inline-amnt`. Also note that if N is not a constant or a constant hole, the result of using this construct will be a set of nested if statements.

There is also a convenient variant of this construct:

```
repeat(i : N){
    stmt
}
```

Where i is a fresh variable name, which will automatically be declared as an integer and can be used within *stmt* to keep track of which copy you are in. The variable will have the value n in the n^{th} copy of *stmt*. As an example of how this may be used, consider the following code:

```
  generator int add([int n, int k], int[n] A, int idx, int[k] offst){
    int res = 0;
    repeat(i: k){
         res += A[idx + offst[i]];
    }
    return res;
  }

  int[n] combine([int n], int[n] A){
        int[n] B;
        for(int i=1; i<n-1; ++i){
             int[3] offsts = {??-1, ??-1, ??-1}; //The -1 are necessary because ?? can only take positive values
```

```
            B[i] = add(A, i, offsts);
        }
        return B;
}

harness void main(){
      assert combine({2, 4, 5}) == {0, 11, 0};
}
```

Note that in the sketch above, the `-1` in the definition of offsts is necessary because constant holes `??` can only take non-negative values. Given the harness, the combine function above will be synthed into the code below.

```
void combine (int n, int[n] A, ref int[n] _out)
{
  _out = ((int[n])0);
  for(int i = 1; i < (n - 1); i = i + 1)
  {
    int _out_s3 = A[i];
    _out_s3 = _out_s3 + (A[i + -1]);
    _out_s3 = _out_s3 + (A[i + 1]);
    _out[i] = _out_s3;
  }
  return;
}
```

Note how the repeat is expanded into three separate statements that read from `A` and add the result to the output variable. The compiler is able to tell that the first one is adding to zero, so it simplifies it and folds it into the declaration of the variable.

5 Regression tests and Benchmark Suite

The sketch distribution includes a set of regression tests that exercise the different corner cases of the language and is important if you are making modifications to the compiler. The tests can be found in the directory `src/test/sk/seq` if you are using the mercurial distribution, or `test/sk/seq` if you are using the easy-to-install version. After having installed the synthesizer, you can run `make long` or `make` if you want the short version of the test. The main difference between the long and the short tests is that the long tests do code generation and test the generated code on random inputs, whereas the short test only checks that the synthesizer doesn't crash.

The distribution also includes a benchmark suite that you can use to evaluate new synthesis algorithms and compare their effect against the standard sketch distribution. This can be run from the `release_benchmarks` directory (or `src/release_benchmarks`) by running `bash perftest.sh OUTDIR`, where `OUTDIR` is a directory where logs should be written. Running the full benchmark suite takes about a day because every test is run 15 times to gather meaningful statistics, but you can modify the script to make it run faster. Once the benchmark suite is running, you can view relevant statistics by running

`> cat OUTDIR/* | awk -f ../scripts/stats.awk`.

6 Advanced Usage and Diagnostics

6.1 Interpreting Synthesizer Output

You can use the flag `-V n` to set the verbosity level of the synthesizer. You can use this to diagnose problems with your sketch, and to understand why a particular problem takes a long time to synthesize.

The first thing you need to understand about SKETCH is that it works by first guessing a solution to the synthesis problem and then checking it. If the check fails, the system generates a counterexample and then searches for a new solution that works for that counterexample and repeats the process. When you run with `-V 5`, you can see each of these inductive synthesis and checking steps as they happen in real time. The synthesizer will output `BEG CHECK` and `END CHECK` before and after the checking phase respectively, and it will output `BEG FIND` and `END FIND` before and after the inductive synthesis phase. Therefore, if the synthesizer seems to be stuck when solving a problem, you can use this output to tell whether it is having trouble with the synthesis or with the checking phase. This is very important, because there are different strategies you can use to speed up the synthesis or the checking phases of the solver.

If the synthesizer tells you that your sketch has no solution, you can also pass the flag `--debug-cex` to ask the synthesizer to show you the counterexamples it is generating as it tries different solutions. Often, these counterexamples can help you pinpoint corner cases that you failed to consider in your sketch.

Flag `-V` *The verbosity flag takes as argument a verbosity level that can range from 0 (minimal output) to 15 (a lot of debug output everything)*

Flag `--debug-cex` *This flag tells the synthesizer to show you the counterexamples that it generates as it tries to find a solution to your problem. You need to pass verbosity of at least 3 to use this flag (*`-V 3`*).*

6.2 Parallel Solving

When running in parallel mode, the SKETCH synthesizer will launch multiple processes and have each process use a combination of stochastic and symbolic search to find a solution to the synthesis problem. Not all problems will benefit from this style of parallelization, but for those that do, the benefits can be significant.

In general, parallelization will only help speed up the synthesis phase, so if your problem is taking a long time in the checking phase, it will not benefit from parallel solving. Similarly, problems that make extensive use of `minimize` may not benefit much from parallel solving.

6.3 Performance diagnostics

Below we list of a few common issues that cause sketch to either take too long to synthesize or to run out of memory, as well as strategies to check whether your sketch suffers from any of those problems.

Too much inlining/unrolling All loops are unrolled by an amount controlled by the `--bnd-unroll-amnt` flag, and all recursive functions are inlined as explained earlier. When there are multiple nested loops or functions that make multiple recursive calls even a small amount of inlining or unrolling can lead to very large representations. If you run with verbosity > 12, in the backend output you will see a series of lines of the form:

`CREATING` $funName$
`size =` N
`after ba size =` N

Each of these corresponds to a function, with N indicating the number of operations performed by the function. If you see a function with a very large N, on the order of (hundreds of thousands), it generally means that within that function, there is either a generator that is too large (see below), or you have too many loops that are being unrolled too much inside that function.

After the backend reads all the functions, it will start inlining. You will see in the output a series of lines of the form:

```
Inlining functions in the sketch.
inlined n1 new size =s1
inlined n2 new size =s2
...
```

This is the synthesizer telling you how many functions it is inlining and how much its internal representation is growing as it inlines. Again, if the numbers start growing too large (above 200K), it means that there is too much inlining. This can usually be fixed by reducing the degree of inlining with `--bnd-inline-amnt`, or by using parallelism.

Recursive generators are too large or have too much recursion Unlike recursive functions, which are inlined in the backend, recursive generators are inlined in the frontend, so if the problem is too much inlining, it can also manifest itself with very large functions, just like the case of too much loop unrolling mentioned above. A big difference, though, is that large generators also introduce a large number of holes. In the synthesizer output you can find a pair of lines that reads as follows:

```
control_ints = N1     control_bits = N2
inputSize = N_in ctrlSize = N_ctrl
```

The two most important numbers are N_{in} and N_{ctrl}, which determine the total number of bits in the inputs and unknowns respectively. If N_{ctrl} grows beyond a few thousand, you are probably using generators that are too big. Follow the tips on Section 4 on how to write more efficient generator functions. Also, the parallel solver can be very effective when your problem is having generators with too many choices.

The range of some intermediate values is growing too large By default, Sketch uses an internal representation that grows linearly with the maximum range of any integers anywhere in the computation. In some cases, this can lead to the problem becoming too large to fit in memory and become impossible to solve. In the solver output, you will see within each synthesis phase a statistic that says:

```
* TIME TO ADD INPUT :  XX ms
```

If this number starts growing beyond a few hundred milliseconds, or if you see the memory consumption growing significantly after every round of the solution algorithm, this usually indicates that you have intermediate values that are growing too large. In this case, there are two flags that you can use. One is `--bnd-int-range`. This flag imposes a bound on the largest integers that you expect to see for any input during a computation (for inputs that fall within the range prescribed by `--bnd-inbits`).

```
harness void main(int x){
  x = x * ??;
  if(x < 10){
    if(??){
      x = x * x;
    }
    if(??){
      x = x * x * x;
    }
    if(??){
      x = x * x * x;
    }
  }
  assert x < 100;
}
```

For example, consider the simple program above; with 5-bit inputs and integer holes, the sketch above will almost surely run out of memory. That is because even though initially the value of `x` ranges from 0 to 32, in the symbolic representation, `x` could reach a value of `2^180`. However, we can see from the sketch, that on any correct run of the resolved sketch, x cannot be greater than 100. We can tell this to the synthesizer by passing `--bnd-int-range 100` (it is usually a good idea to give it some additional margin, so you could do `--bnd-int-range 110`). With this flag, the sketch above resolves in less than a second.

In some cases, though, the maximum size of your integers is too large, so even with the `--bnd-int-range` flag, it still runs out of memory. In those cases, you can use a flag `--slv-nativeints`. This switches to a completely different solver that scales better to large numbers, but can be slower than the default solver. If the range of integers that are possible for values in your program given inputs within the input range is greater than about 500, then you will probably see a speedup from using this flag.

Flag `--bnd-int-range` *Maximum absolute value of integers modeled by the system. If this is not passed, the maximum value is dictated by* `MAX_INT`*, but you are likely to see major scalability problems long before your values reach that size.*

Flag `--slv-nativeints` *Replaces the standard solver with one that is more memory efficient, especially for problems involving larger integer values.*

The problem is too hard to verify The sketch synthesizer iterates between synthesis and checking phases. When running with high verbosity, the synthesizer prints `BEG CHECK` and `END CHECK` at the beginning and the end of the checking phase. It is common to have situations where the synthesizer can easily find a correct solution, but finds it hard to prove it correct, even for the input bound provided. There are a few things that can be done to remedy this. First, there is a flag `--slv-lightverif`, which tells the solver to make only a best effort in trying to check the solution. If the solver fails to find a counterexample in the allotted time, it assumes that the program is correct.

Another option is to run with a smaller input range, either fewer bits of input (`--bnd-inbits`) or smaller maximum size of arrays for sketches that have array inputs (`--bnd-arr-size`). In some cases, though, smaller sizes may not be enough to force the synthesizer to provide a correct answer. One option in such cases is to provide one test harness that checks for all small inputs, and then a separate harness that checks for against some explicitly chosen large inputs, rather than trying to check against all large inputs. For example, consider the code below.

```
int foo(int x){
  x = x + ??;
  while({| x (> | >=) 1000 |} ){
    x = {| x (- | + ) 1000 |};
  }
  return x;
}
harness void main(int x){
  assert ((x + 5) % 1000) == foo(x);
}
```

In order to exercise the case inside the while loop, you would need `--bnd-inbits` of at least 10, on the other hand, you can provide a smaller bitwidth for the input, and then provide an explicit harness that tests this for a few carefully selected larger numbers, or even a range of larger numbers. For example, if in addition to the main harness above, I provide the `largeVal` harness below and I set `--bnd-inbits 5`, `main` will check against values in the range $[0, 31]$, and `largeVal` will check against values in the range $[1090, 1121]$.

```
harness void largeVal(int x){
  main(1090 + x);
}
```

Note that if I had only hard-coded a few values in `largeVal`, I could have missed the crucial value of 1095, which is essential to ensuring that `foo` is synthesized to the correct `x>=0` instead of the incorrect `x>0`.

For sketches that contain arrays, often checking against all possible values for all entries in the array is overkill. In such cases, you can use the option `--slv-sparsearray`x, which causes the solver to verify only against sparse arrays, i.e. arrays that have mostly zeros. x is the degree of sparsity. So $x = 0.05$ for an array of size 1000 means that the checker will only check against inputs that have at most 50 non-zero entries. This can be especially useful if your sketch takes multi-dimensional arrays as inputs.

Finally, the checker within sketch uses a combination of random testing and symbolic checking in order to find errors in the candidate programs. If you use flags such as `--slv-lightverif` which reduce the effectiveness of symbolic checking it is advised that you increase the amount of random testing. You can do this by using the flag `--slv-simiters`N, where N controls the amount of effort to spend on random testing. The default is 3, but values as high as 150 can be useful for some benchmarks. You will also note that as it performs random testing, it will also periodically use insights it learns through random testing to attempt to simplify the verification problem, so higher values of N can lead to further simplification, which can sometimes eliminate the need to use `--slv-lightverif`.

6.4 Custom Code Generators

For many applications, the user's goal is not to generate C code, but instead to derive code details that will later be used by other applications. In order to simplify this process, SKETCH makes it easy to create custom code generators that will be invoked by the sketch compiler at code generation time.

Custom code generators must implement the `FEVisitor` interface defined in the `sketch.compiler.ast.core` package and must have a default constructor that the compiler can use to instantiate them. In order to ask the compiler to use a custom code generator, you must label your custom code generator with the `@CodeGenerator` annotation. You must then package your code generator together with any additional classes it uses into a single jar file, and you must tell SKETCH to use this jar file by using the flag `--fe-custom-codegen`.

Flag `--fe-custom-codegen` *Flag takes as an argument the name of a jar file and forces* SKETCH *to use the first code generator it finds in that file.*

To illustrate how to create a custom code generator, the SKETCH distribution includes a folder called `sketch-frontend/customcodegen` that contains a custom code generator called `SCP` that simply pretty-prints the program to the terminal. In order to get SKETCH to use this class as a code generator, follow these simple steps:

- From the `sketch-frontend` directory, compile the code generator by running
  ```
  > javac -cp sketch-1.7.6-noarch.jar customcodegen/SCP.java
  ```
- Create a jar file by running
  ```
  > jar -cvf customcodegen.jar customcodegen/
  ```
- Try out your new code generator by running
  ```
  > sketch --fe-custom-codegen customcodegen.jar test/sk/seq/miniTest1.sk
  ```

When you run, you should see the following messages in the output:

```
Class customcodegen.SCP is a code generator.
Generating code with customcodegen.SCP
(followed by the pretty-printed version of your code).
```

6.5 Temporary Files and Frontend Backend Communication

The sketch frontend communicates with the solver through temporary files. By default, these files are named after the sketch you are solving and are placed in your temporary directory and deleted right afterwards.

One unfortunate consequence of this is that if you run two instances of sketch at the same time on the same sketch (or on two sketch files with the same name), the temporary file can get corrupted, leading to a compiler crash. In order to avoid this problem, you can use the flag `--fe-output` to direct the frontend to put the temporary files in a different directory.

Flag `--fe-output` *Temporary output directory used to communicate with backend solver.*
Also, if you are doing advanced development on the system, you will sometimes want to keep the temporary files from being deleted. You can do this by using the `--fe-keep-tmp` flag.

Flag `--fe-keep-tmp` *Keep intermediate files used by the sketch frontend to communicate with the solver.*
In some cases, especially when debugging sketches that take a long time to solve, you may find it useful to not have to rerun the solver after having made small changes to your sketch. Sketch provides a flag `--debug-fake-solver`, which tells the system to not invoke the backend solver; instead, if the system can find an existing temporary file with a solution for this sketch, it will just use those results. For example, if you have a sketch `slow.sk` which takes a long time to run, you can invoke:

```
> sketch --fe-keep-tmp slow.sk          // This runs for a long time but keeps around the temporary files.
> sketch --debug-fake-solver slow.sk   // This runs very fast because it doesn't actually try
                                                    // to solve the sketch, just reuses the results from the last
```

You should be careful when using this flag, because if you changed your sketch significantly from one run to the next, the results from the old sketch may not be correct for the new sketch.

Flag `--debug-fake-solver` *Instead of invoking the solver, sketch searches for an existing intermediate file from a prior run and uses the results stored in that file.*

6.6 Extracting the intermediate representation

If you have your own SMT solver with support for quantifiers and you want to compare your performance with Sketch, you can ask the solver for the intermediate representation of the synthesis problem after it is done optimizing and desugaring the high-level language features.

Flag `--debug-output-dag` *This flag outputs the intermediate representation in an easy to parse (although not necessarily easy to read) format suitable for mechanical conversion into other solver formats. The flag takes as a parameter the file name to which to write the output.*
The file will show all the nodes in the intermediate representation in topological order. There listing in Figure 1 shows all the different types of nodes and the format in which they are written.

```
id = ARR_R      TYPE    index   inputarr
id = ARR_W      TYPE    index   old-array       new-value
id = ARR_CREATE TYPE    size    v0 v1 ....
id = BINOP      TYPE    left    right
        // where BINOP can be AND, OR, XOR, PLUS, TIMES, DIV, MOD, LT, EQ
id = UNOP       TYPE    parent  // where UNOP can be NOT or NEG
id = SRC        TYPE    NAME    bits
id = CTRL       TYPE    NAME    bits
id = DST        TYPE    NAME    val
id = UFUN       TYPE    NAME    OUT_NAME CALLID ( (size p1 p2 ...) | (***) )
id = ARRACC     TYPE    index   size    v0 v1 ...
id = CONST      TYPE    val
id = ARRASS     TYPE    val == c noval yesval
id = ACTRL      TYPE    nbits b0 b1 b2 ...
id = ASSERT     val     "msg"
```

Figure 1: Format for intermediate representation.

7 Standard Library

7.1 Package math

This package includes standard math functions. These functions only work with the floating point encoding `--fe-fpencoding TO_BACKEND`, so including this package automatically sets that flag.

While the functions in this package look like uninterpreted functions, the backend actually understands their semantics. `@Native` annotations to generate the proper code during code generation.
Location : math.skh

7.1.1 Functions

- `uninterp float sin (float v)/*math.skh:15*/`
- `uninterp float cos (float v)/*math.skh:17*/`
- `uninterp float tan (float v)/*math.skh:19*/`
- `uninterp float sqrt (float v)/*math.skh:21*/`
- `float abs (float v)/*math.skh:23*/`
- `uninterp float log (float v)/*math.skh:28*/`
- `uninterp float arctan (float v)/*math.skh:33*/`
- `uninterp float _cast_int_float (int v)/*math.skh:40*/`

  Overrides the default cast from floats to ints. This looks like an uninterpreted function, but is actually an intrinsic that has special meaning for the backend.

- **generator double** `_cast_int_double (`**int** `v)/*math.skh:46*/`

  Overrides the default cast from doubles to ints.

- `uninterp` **float** `exp (`**float** `v)/*math.skh:51*/`

- **generator float** `Pi ()/*math.skh:55*/`

7.2 Package generics

This package includes commonly used generic functions. All the functions in this package are generators, so they get inlined into their calling context.
Location : generics.skh

7.2.1 Functions

- **generator** `T[n] map <T> ([`**int**` n], T[n] in, fun f)/*generics.skh:17*/`

  Standard map function applies a function to every element in an array and produces a new array. The function parameter `f` is assumed to be of type `T->T`.

- **generator int** `len <T> ([`**int**` n], T[n] ar)/*generics.skh:29*/`

  Returns the length of an array. Useful in the context of array constructors and string constants.

- **generator void foreach** `<T> ([`**int**` n], ref T[n] in, fun f)/*generics.skh:38*/`

  Apply a function to every element of an array. The function parameter `f` is assumed to be of type `T->`**void**.

- **generator void** `forall (`**int**` a, `**int**` b, fun f)/*generics.skh:49*/`

  Apply a function to all integers between a and b. The function parameter `f` is assumed to be of type **int**`->`**void**.

7.3 Package generators

Generator functions for arithmetic operations.
Location : generators.skh

7.3.1 Functions

- `generator int linexp ([int N], int[N] vals)/*generators.skh:19*/`

  Linear expression generator with coefficients between -2 and $(2^{cbits} - 2)$

- `generator int plinexp ([int N], int[N] vals)/*generators.skh:35*/`

  Linear expression generator with coefficients between 0 and (2^{cbits})

- `generator int expr ([int N, int T], int[N] pars, int[T] ops)/*generators.skh:51*/`

  Generate an integer expression based on the N operands and T operators.

- `generator int exprG ([int T], fun chose, int[T] ops)/*generators.skh:68*/`

  Generate an integer expression based on choices provided by a generator chose and T operators. Chose should take a reference integer parameter and should set it to the choice id it picked. This will help the generator avoid symmetries.

- `generator int expr_cost ([int N, int T], int[N] pars, int[T] ops, ref int cost)/*generators.skh:83*/`

  Generate an expression based on the N operands and T operators. The ref parameter cost will be set to a cost that is proportional to the size of the expression.

- `generator int expr_noc_cost ([int N], int[N] pars, int T, int[T] ops, ref int cost)/*generators.skh:98*/`

  Generate an expression based on the N operands and T operators given.

- `generator bit exprBool ([int N, int T], int[N] pars, int[T] ops)/*generators.skh:117*/`

  Generate a boolean expression based on the N operands and T operators given as arrays. The operators are arithmetic expressions and the boolean expression will be produced by generating boolean combinations of inequalities built from those arithmetic expressions.

- `generator bit exprBoolG ([int T], fun choices, int[T] ops)/*generators.skh:126*/`

7.4 Package array

Array library includes both fixed size and variable size arrays. The main reason to use these instead of the built in array types is if you do not want the array length to be part of the type.
Location : array.skh

7.4.1 Datastructures

Structure SArray

```
struct SArray<T> {
    int n;
    T[n] val;
}
```

Static sized array. The main reason to use this object instead of just an array of size T[n] is if you do not want the type to include the array size. For example, suppose you have a filter function `filter<T>(int n, T[n] in, fun pred)` that takes an array as input and filters out all elements on which predicate `pred` returns **`false`**. We do not know how many elements the output is going to have, so the return type cannot just be int[q]. Instead, the return type can be `SArray<T>`.

Structure Array

```
struct Array<T> {
    SArray<T> inner;
    int sz;
}
```

This is a variable size array, similar to a `Vector` datatype in other languages.

7.4.2 Functions

- `Array<T> newArray <T> ()/*array.skh:33*/`

  Create a new empty array of length zero.

- `T last <T> (Array<T> in)/*array.skh:40*/`

  Return the last element of the array `in`.

- `void add <T> (Array<T> in, T val)/*array.skh:48*/`

  Add an element `val` at the end of array `in`.

7.5 Package list

List library includes a list ADT and a set of helper functions to work with it.
Location : list.skh

7.5.1 Datastructures

Structure List

```
struct List<T> {
    @Immutable()
}
```

ADT based list.

Structure Cons

```
struct Cons<T> extends List@list {
    T val;
    List<T> next;
}
```

Structure Nil

```
struct Nil<T> extends List@list {
}
```

7.5.2 Functions

- `List<T> empty <T> ()/*list.skh:20*/`

  Returns an empty list.

- `List<T> single <T> (T x)/*list.skh:28*/`

  Constructor for a list with a single element x.

- `T head <T> (List<T> l)/*list.skh:35*/`

  Return the first element on a list; requires the list to be non-empty.

- `List<T> add <T> (List<T> lst, T val)/*list.skh:42*/`

  Adds an element to the beginning of a list.

7.6 Package stack

Immutable generic stack datatype.
Location : stack.skh

7.6.1 Datastructures

Structure Stack

```
struct Stack<T> {
    @Immutable()
}
```

Abstract immutable Stack type. You should only construct this type using the `Empty` constructor listed below.

7.6.2 Functions

- `Stack<T> Empty <T> ()/*stack.skh:20*/`

  Creates an empty stack.

- `Stack<T> push <T> (Stack<T> prev, T val)/*stack.skh:27*/`

  Pushes an element into the stack and returns the new stack.

- `Stack<T> pop <T> (Stack<T> st)/*stack.skh:34*/`

  Pops the top of the stack and returns the new stack.

- `T peek <T> (Stack<T> st)/*stack.skh:42*/`

  Returns the value of the top of the stack.

8 Credits

The sketch project was started at UC Berkeley in 2005 by Armando Solar-Lezama and Ras Bodik and has been led by Solar-Lezama at MIT since 2009. The current code base includes important contributions by the following individuals (in chronological order): Gilad Arnold, Liviu Tancau, Chris Jones, Nicholas Tung, Lexin Shan, Jean Yang, Rishabh Singh, Zhilei Xu, Rohit Singh, Jeevana Priya Inala, Xiaokang Qui, Miguel Velez. The project also relies heavily on code from MiniSat (Niklas Een, Niklas Sorensson), StreamIt (led by Bill Theis and Saman Amarasinghe with code from David Maze, Michal Karczmarek and others), as well as the open source systems ANTLR (Terence Parr), Apache Commons CLI and Rats/xtc (Robert Grimm).

Over the years, the project has benefited from funding by the following projects:

- NSF-1049406 EAGER:Human-Centered Software Synthesis
- NSF-1116362 SHF: Small: Human-Centered Software Synthesis
- NSF-1161775 SHF: Medium: Collaborative Research: Marrying program analysis and numerical Search
- DOE: ER25998/DE-SC0005372: Software Synthesis for High Productivity Exascale Computing
- NSF-1139056 Collaborative Research: Expeditions in Computer Augmented Programming
- DARPA: UHPC Program

- DOE: ER26116/DE-SC0008923: D-TEC: DSL Technology for Exascale Computing
- DARPA: MUSE FA8750-14-2-0242
- Toyota Research Institute

9 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 $\texttt{bnd-arr-size}^2$.. 9

`--bnd-bound-mode` The solver supports two bound modes: `CALLSITE` and `CALLNAME`. In `CALLNAME` mode (the default), the flag `bnd-inline-amnt` will bound the number of times any function appears in the stack. In the `CALLSITE` mode, the `bnd-inline-amnt` flag will bound the number of times a given *call site* appears on the stack, so if the same function is called recursively multiple times, each site is counted independently.. 17

`--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.. 31

`--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.. 3

`--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.. 17

`--bnd-int-range` Maximum absolute value of integers modeled by the system. If this is not passed, the maximum value is dictated by `MAX_INT`, but you are likely to see major scalability problems long before your values reach that size.. 41

`--bnd-mbits` The flag `bnd-mbits` tells the synthesizer how many bits to use to represent all bounds introduced by `minimize(e)`(default 5). Note that the largest value of `(e)` will be less than the bound, so if `e` can have value n, the bound needs enough bits to be able to reach $n+1$.. 31

`--bnd-unroll-amnt` This flag controls the degree of unrolling for both loops and **`repeat`** constructs. 16

`--debug-cex` This flag tells the synthesizer to show you the counterexamples that it generates as it tries to find a solution to your problem. You need to pass verbosity of at least 3 to use this flag (`-V 3`).. 39

`--debug-fake-solver` Instead of invoking the solver, sketch searches for an existing intermediate file from a prior run and uses the results stored in that file.. 43

`--debug-output-dag` This flag outputs the intermediate representation in an easy to parse (although not necessarily easy to read) format suitable for mechanical conversion into other solver formats. The flag takes as a parameter the file name to which to write the output.. 43

`--fe-custom-codegen` Flag takes as an argument the name of a jar file and forces SKETCH to use the first code generator it finds in that file.. 42

`--fe-fpencoding` This flag controls which of three possible encodings are used for floating point values. `AS_BIT` encodes floating point values using a single bit; addition and subtraction are replaced with `xor`, and multiplication is replaced with `and`. Division and comparisons are not supported in this representation, nor are casts to and from integers. `AS_FFIELD` will encode floating points using a finite field of integers mod 7. This representation does support division, but not comparisons or casts. Finally, `AS_FIXPOINT` represents floats as fixed point values; this representation supports all the operations, but it is expensive. Finally, the `TO_BACKEND` representation uses the floating point support in the backend solver. It is the most expressive representation, and is set by default when you include the `math.skh` library. Its cost is very problem specific; for some problems it can be very efficient, but it can also exhibit poor scalability.. 4

`--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.. 24

`--fe-keep-tmp` Keep intermediate files used by the sketch frontend to communicate with the solver.. 43

`--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. 3

`--fe-output-test` This flag causes the synthesizer to produce a test harness to run the C++ code on a set of random inputs.. 3

`--fe-output` Temporary output directory used to communicate with backend solver.. 43

`--slv-nativeints` Replaces the standard solver with one that is more memory efficient, especially for problems involving larger integer values.. 41

`--slv-p-cpus` This flag can be used in combination to the `--slv-parallel` flag to indicate to the synthesizer how many cores to use.. 4

`--slv-parallel` This flags enables parallel mode, allowing the synthesizer to take advantage of multiple cores. By default, the synthesizer will use one less core than the total number of cores available in your system. 4

`-V` The verbosity flag takes as argument a verbosity level that can range from 0 (minimal output) to 15 (a lot of debug output everything). 39