The Sketch Programmers Manual

For Sketch Version 1.6.4

Contents

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* $2^{\texttt{bnd-inbits}}-1$*. The default for this flag is 5; attempting numbers much bigger than this is not recommended.*

1.2 Running the synthesizer

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

```
> sketch   test1.sk
```

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

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

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

Flags can be passed to the compiler in two ways. The first and most traditional one is by passing them in the command line. For the example above, you can get code generated by invoking the compiler as follows.

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

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

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

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

2 Core language

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

2.1 Primitive Types

The sketch language contains 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 Floating point values (either **float** or **double**) are not handled natively by the synthesizer, so they have to be modeled using other mechanisms. The sketch synthesizer currently supports three different encodings for floating point values, which can be controlled by the flag `--fe-fpencoding`.

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 the most expensive.*

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 p1 = new Point(x=5, y=7);
  assert p1.x == 5 && p1.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. Also, just like in Java, all pointer dereferences have an implicit null pointer check.

2.3 Temporary Structures

There are instances where it's 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 don't 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 assumption that the reference will not be null. Failure to satisfy the assumption will cause an exception.

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 lists 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 2.9) .
- 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 2.9) 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<N; ++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}.

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`$\leq$`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`$\geq$`N`. If `M`$\neq$`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(T),...,pad(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 three 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.

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 Control Flow

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

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

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

Example 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. For the third loop, the unroll factor will be controlled by the flag, but will never exceed seven.*

2.9 Functions

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

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

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

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

Reference Vs. Value Parameter Passing By default, parameter passing is done by value, 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==n
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
```

2.10 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.

2.11 Local functions and closures

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

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

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

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

2.12 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.

2.13 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 21. *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
```

2.14 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.

2.15 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.

The synthesizer currently supports two annotations. 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 22. *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.

3 Constant Generators and Specs

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

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

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

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

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

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

3.1 Types for Constant Generators

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

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

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

3.2 Ranges for holes

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

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

3.3 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.

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.

```
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);
}
```

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 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);
    }
}
```

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 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.

7 Advanced Usage and Diagnostics

7.1 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.

```
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.

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.*

8 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 $\texttt{bnd-arr-size}^2$.. 6

`--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.. 16

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

`--bnd-inline-amnt` Bounds the amount of inlining for any function call. The value of this parameter corresponds to the maximum number of times any function can appear in the stack.. 10

`--bnd-unroll-amnt` This flag controls the degree of unrolling for both loops and **`repeat`** constructs. 10

`--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.. 20

--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 the most expensive.. 2

--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.. 12

--fe-keep-tmp Keep intermediate files used by the sketch frontend to communicate with the solver.. 21

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

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

--fe-output Temporary output directory used to communicate with backend solver.. 21