Virtual eXecutor Compiler

The compilation is split into seperate phases. Listed in executing order.

- Weeding
  - Constant values. Syntactical checks and transformations
- Static type checking
  - cheching for size and sign
- Environment Building
  - calculate all constant expressions
  - build scoped environment
  - subexpression elimination
- Liveness analysis
  - calculate stack segment
- Code Generation
  - apply patterns, insert values from env.

1 Weeding

1.1 Decorations

The weeder will fill out the values of all constants in the source program. It will also annotate every constant expression with its precalculated static value if applicable.

- For each integer literal we
  - check that it is a legal 32 bit signed integer.
  - find and annotate the expression with the smallest type it will fit into.
  - annotate the expression with that value in the environment.
- Chars
- Strings
- Floats
- Subexpression elimination ?

1.2 Checks

The weeder will do the following syntactical checks:

- For every port declaration we check that the address expression evaluates to a *IntConstExp*.
- Every init expression must be *AinitExp*. *(needed?)*
- A *PType* node that is not the return value of a function, must not be *AVoidType*.

2 Environment Building

The environments phase builds maps from identifiers to declarations for all declared entities in the program. The two constructs that controls the scoping of the environment are Functions and blocks.

2.1 Decorations

- Add all variable declarations to the current scope
- Add all port declarations to the current scope
- Add all function declarations to the current scope
- Add all formal parameters to the scope of their contained function.

2.2 Checks

- Check that no two variables in the same scope has identical names. (shadowing should work.)
- Check that no two ports *in visible scope* has identical names.
  (no overloading).
- Check that no functions *in visible scope* has identical name.
  (no overloading)
- Check that the init expression of a variable declaration resolves to a

3 Type Checking

4 Grammar

Basic syntactic sets

integers = N	*ranged over by int*
chars = {'a',..'Z'}	*ranged over by char*
truthvalues = {True, False}	*ranged over by bool*
*strings = {char }**	*ranged over by string*
constants = {strings} U {integers}	*ranged over by const*

Declarations

decl ::= var_decl |port_decl | func_decl
var_decl ::= sign type size identifier [init]:exp
port_decl ::= type size identifier

Statements

stm ::= decl |if *exp* then *stm* |if *exp* then *stm* else *stm'* |*stm* ; *stm'* |
for (*exp* ; *exp'* ; *exp") stm* |return *exp* |break .

Expressions

exp ::= var |const |cond_exp | arit_exp .

arit_exp ::= *exp* arit_unop |*exp* arit_binop *exp* .
arit_unop ::= inc |dec |sizeof |negate .
arit_binop := assign |plus |minus |mult |div |mod |lshift |rshift |
bit_xor |bit_and |bit_or .

cond_exp ::= *exp* cond_unop |*exp* cond_binop *exp* .
cond_unop ::= not .
cond_binop ::= and |or |eq |not eq |lt |lteq |gt |gteq |cond .

5 Semantics

Type System

types = *{single, double, quad, bool}*   *ranged over by type.*
tenv : (*vars* U *ports*) --> *type*   *identified by T.*

Dynamic

Inference rules for determining expresssion well formedness

```
              T|exp : t
[var_decl]    ----------------------------------------------   iff size(t) <= size(type) thenT2 = T[id/type].
              T|type id = exp --> T2
```

Static type check of expressions

type relation : *tenv X exp X type*   *notation: T |- exp : t.*
"given static type env T the expression exp yields the type t".

```
[const]              ------------------------ type = the smallest type const may fit into.
                      T |- const: type

[var]                ---------------t = T(var)
                      T |- var : t

                      T |- exp : t
[unop__inc]          --------------------- iff t != bool
                      T |- exp++  :  t

                      T |- exp : t
[unop__dec]          ---------------------- iff t != bool
                      T |- exp--  :  t

                      T |- exp : t
[unop__size]         ------------------------------   iff t != bool
                      T |- sizeof(exp) : single

                      T |- exp :  t
[unop__neg]          ----------------------
                      T |- neg exp :  t

                     T |- exp: t
[unop__not]          ------------------------
                     T|-  not exp : t

                       T|exp : t       T|exp* : t'
[binop__assign]      --------------------------------- iff sizeoff(t') <= sizeoff(t)
```

```
                         T|exp assign exp* :  void

[binop__plus]            T|exp: t1     T|exp': t2
                         ------------------------------  t = max( t1, t2 )
                          T| exp plus exp' : t
```

Note maybe add a check to see if values could be statiaclly determind and thus providing for over/under flow coverage (auto upcast).

do well formedness
- correct shadowing of variables
- correct function overloading
- expressions are type correct (ie. bool vs numerial)

type checking
- check for narrowing conversions
- check for divide by zero

assign all simple expressions to values.
weed away any sub expression that have no named references and replace it with a constant.

Dynamical type resolving:

exp ::= named |short const |int const |wide const |exp unop |exp binop exp'

Transition system (expressions)

TTS = (S, T, tfunc)
S = tenv X exp U (valuefork U signfork)
T = signfork
tfunc : S --> S

define runtime environment:
renv : { *var* U *port* } --> *value*

Variables :

Their bit length are statiaclly assigned. This cannot change! Although a smaller dimension can always be copied into a larger dimension. Thus single can evolve larger types if need be. Static type checking reports an error if an assignment of a larger type into a smaller occurs. The sign of a variable is determined at runtime. It is modelled as a fork that can be evaluated to both representations on demand.

Ports :

Are much like variables except I assume you cannot write a signed value to an unsinged port without excepting weirdness. However the interpretation of a read is up to the L-value.

VM

signed / unsigned fork handling (runtime)

execeptions and try catches.

Memory model

Ports

The IO dimension of VXC lies solely in the hands of the port declaration.
It supports fast buffered read and write operations in custom allocateable blocks.
These are typically implemented using scope pointers. There are two dimensions of the semantics of a port. The first is just *dimension*, which determins the size of the associated port buffer. The second is *stream ordering* which acts as a protocol for the read and write operations of the port.
It is worth while to notice that all expressions in source code that does not *read from* a port must! be constant and there for resolveable! Loops might kill this idea a bit.

Scope

The scope construct is a big thing in this language. On entering a scope all the associated memory will be allocated and initialized. When exiting it is GC'ed.
The global scope is the module scope. At Runtime when ever the program counter exits a function, local or global scope certain flags are set and the runtime can react to this. There is another dimension to this. It is called memory scope. This is also managed by the runtime, and essentially does range checks to assert that no pointer or loop *ever* exceeds its predetermined boundary conditions. It too can raise flags for the runtime to react.
The language supports two kinds of type safe iterators over ordered *statically declared* variables. Any variable is essentially a buffer of a *distinct* type and a dimension. The default dimension is one. The foreach loop can in most cases statically assert that the operation is safe, because it controls the scoped pointer.

Scope guarded memory
Every value that is reachable and referenced in the scope must them selves be scope guarded. Their values are copied to the local scope. You can delcare pointers to any cell within the local scope or to the scope itself. Typed Iterators are available, and they are completely safe. When the memory scope exits a flag is raised that acts like a legal break operation and the scope pointer is reset to 0.

Dynamic memory
All memory declared based on some expression that somewhere does a read from a port or has a undecideable loop construct is considered dynamic. In short: Our analysis cannot statically figure out how much memory to allocate. This memory is accessible only by value and is placed normally onto the stack of the current scope.

Scope pointers
All variable declarations in local scope are seen as one static memory allocation. That is referenceable and iterateable. This is called a scope pointer and it is linked to array indexer behaviour. Suppose you define a memory layout like this:

```
single head;
double adress;
quad[2] data;
single tail;
```

This yields the following ordered memory layout of 12 bytes.

```
head:      [0   ]
adress:    [1   ][2   ]
data:      [3   ][4   ][5   ][6   ]
           [7   ][8   ][9   ][10  ]
tail:      [11  ] .
```

If you instead went:

```
single[12] data;
```

you would get 12 ordered consequtive bytes, while

```
quad[255] data;
```

gives you 255 * 4 bytes of ordered data. The sole purpose of the ordering of data is to automate the iteration and stream operations over the data. You could always just use single arrays if you wanted a 1:1 relationship between indice and byte count.

Pointers

there are two kinds of pointers. They both point to local scope memory adresses. You cannot reference or modify memory from other scopes via them. All values are copied to your own scope upon read requests. Assignment for pointers depends on the type of the lvalue. If it is a pointer of compatible it will be set to point to the same address as the rvalue. If the lvalue is a local variable or port the data at the pointer adress will be assigned to the lvalue (if applicable).

scope pointers iterate over typed information, expressions using the values of the pointers can be statically checked for precision loss.

```
scope* p = this;
foreach ( c , p , p++)
{ ... }
```

data pointers iterate over untyped information, iterpreting the data by its declared size

```
single* ps = this;
```

```
foreach ( c  , ps , ps++)
{ ... }
```

p will be (0, 1, 3, 11) and ps will be (0,1,2,3,...,11). Because the memory is statically allocated it can be handled that the memory pointer never goes of out scope unhandleded, and that the interpretation is done with out error for loss of precision.

Bitwise control

for normal expressions shift operations works as usual. For port declarations they could act as strong type safe stream operator.

```
port char_buffer : 3;
scope* p = this;
p[0] = char_buffer<<8     //consumes one byte from the buffer
p[1] = char_buffer<<16    //consumes two bytes from the buffer
p[2] = char_buffer<<64   //consumes eight bytes from the buffer
p[3] = char_buffer<<8     //consumes one byte from the buffer
char_buffer>>96 = p[0]   //writes the entire thing back to the buffer.

single x = p;
```

Virtual Machine runtime flags

Scope guards

flags should be raised whenever a memory scope exists so we dynamically can break foreach loops on a memory pointers to make them safe. A transaction mechanismn can also be built with scope guards.

Overflow /Sign mismatch

flags the check for flow errors should be set.

IO Port failure

flags that signal the error of a particular IO device should be set.

Out of bounds

flags that signal out of boundary request on array index expressions should be set.

```
struct packet {
quad address;
quad server_response;
double crc;
quad[16] data;
}

port quad in : 0;
port quad out : 0;
packet* pInPacket;
packet* pOutPacket;

quad[16] peerIn( )
{
        try {
          networkIn( );
          protocolIn( );
          applicationIn( );
          return pInPacket[3];
        }
        catch {
        }
}

void peerOut(quad[16] data, double address, double repsonse)
{
          try {
                    data >> mpOutPacket[3];
                    calcCrc16(data) >>mpOutPacket[2] ;
                    response >> mpOutPacket[1]
                    address >> mpOutPacket[0];
                    mpOutPacket>>out;
          }
          catch {
          }
}

void networkIn()
{
          in>>pInPacket[0];
          in>>pInPacket[1];
          if (pInPacket[0] == ”127.0.0.1” && pInPacket[1] == ”RESPONSE_OK”)
          {
                    in>>pInPacket[2];
```

```
		in>>pInPacket[3];
	}
}

void protocolIn()
{
	double crc = calcCrc16(pInPacket[3]);
	if (crc != pInPacket[2])
	{
		throw new Exception(”Crc check failed”);
	}
}
```