Implementing a Domain Specific Language for Network Drivers

Russell Yanofsky – rey4@columbia.edu

Basic Idea: Reimplementation of Devil Compiler as a C++ Framework

- Devil = DEVice Interface Language, a Domain Specific Language (DSL) used to help write driver code.
    - Input: Description of hardware using Devil syntax
    - Output: C functions that let programmer access hardware in a high level manner.

- My Library, a C++ Framework will provide template classes which implement same functionality as Devil. Advantages:
    - No separate executable tools needed, just a standard C++ compiler
    - Should be more easier to extend with plug in classes.

Driver Code Fragment for Logitech Mouse

```
#define MSE_DATA_PORT 0x23c
#define MSE_SIGNATURE_PORT 0x23d
#define MSE_CONTROL_PORT 0x23e
#define MSE_READ_Y_LOW 0x80
#define MSE_READ_Y_HIGH 0xa0

outb(MSE_READ_Y_LOW, MSE_CONTROL_PORT );
dy = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
buttons = inb(MSE_DATA_PORT);
dy |= (buttons & 0xf) << 4;
```

Example originally appeared on this paper: Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, Gilles Muller. Devil: An IDL for Hardware Programming. OSDI 2000, pages 17-30, San Diego, October 2000.

Same behavior written in Devil Interface Language:

```
// device
device logitech_busmouse (base : bit[8]
  port @ {0..3})
{
  // index register
  register index_reg = write base @ 2,
    mask '1..00000' : bit[8];
  variable index = index_reg[6..5] : int(2);

  // registers for low and high bits
  register y_low  = read base @ 0, pre
    {index = 2} : bit[8], mask '****....';
  register y_high = read base @ 0, pre
    {index = 3} : bit[8], mask '...*....';

  // dy variable
  variable dy = y_high[3..0] # y_low[3..0],
    volatile : signed int(8);
}

dy = get_dy();
```

Same behavior written with a C++ library.

```
typedef Register<2, 8,
  List<ReadOnly, Mask... > > IndexReg;

typedef Variable<2, AtRegister<IndexReg, 6, 5>
  > Index;

/* Base Address, Index Register, Index Value, Bit Size, Constraint
typedef IndexedRegister<0, Index, 2, 8,
  Mask... > > Y_Low;
typedef IndexedRegister<0, Index, 3, 8,
  Mask... > > Y_High;

typedef Variable<8, List<
  AtRegister<Y_Low, 3, 0>,
  AtRegister<Y_High, 3, 0> > > DY;

y += dy;
```

C++ Overloaded operators allow arbitrary actions to be performed when a variable is read from or written to.

```
template<...>
class Variable
{
  Variable & operator=(int rvalue)
  {
    ... executed whenever variable is assigned an int ...
  }

  operator int()
  {
    ... executed when variable is read from ...
  }
};
```

Typelist primitives:

```
struct Null;

template<typename T, typename NEXT>
struct Node
{
  typedef T type;
  typedef NEXT next;
};
```

Length 3 typelist built out of primitives:

```
typedef Node<int, Node<signed int,
  Node<unsigned int, Null> > >  MyList;
```

Same typelist with library-provided shorthand:

```
typedef List<int, signed int, unsigned int> MyList;
```

An example typelist Algorithm.

```
template<typename LIST>
struct Length;

// Length of a Null typelist is 0
template<>
struct Length<Null>
{
  enum { value = 0 };
};

// Length of at a Typelist node is 1 + Length at Next Node
template<class T, class U>
struct Length< TypeList<T, U> >
{
  enum { value = 1 + Length<U> };
};

// How to access the length
cout << Length<MyList>::value;
```

"Curiously Recursive Template Pattern"

```
template<class T_leaftype>
class Matrix {
public:
  T_leaftype& asLeaf()
  { return static_cast<T_leaftype&>(*this); }

  double operator()(int i, int j)
  { return asLeaf()(i,j); }
};

class SymmetricMatrix :
  public Matrix<SymmetricMatrix> {
  ...
};
```