Implementing a Domain Specific Language for Network Drivers

Russell Yanofsky – rey4@columbia.edu
Department of Computer Science, Columbia University

Abstract – **This report consists of an overview of previously described and implemented Domain Specific Languages (DSLs) for writing device drivers. It discusses some of the benefits and drawbacks of these approaches and shows how a C++ library can be used instead of a standalone language to achieve many of the same goals.**

I. Introduction

Network device drivers are essentially glue code that pass instructions and data between a low-level network hardware layer and a higher level networking interface determined by the operating system. Network drivers for a given operating system tend to have a lot of code in common. For one thing, their structure will be similar, because they all implement the same packet delivery API. Moreover, they all use similar bit-manipulation techniques in order to send and receive information from hardware registers. These pieces of code which device drivers have in common, asynchronous communication and register oriented bit manipulation, happen to be things that are difficult to express in a generic way in C, the language in which most device drivers are written. Domain Specific Languages offer a way to write device driver code in a more expressive way. Domain Specific Languages (DSLs) designed for use in network drivers, or for hardware drivers in general allow common code to be factored out, and low-level code to be automatically generated, so that driver writers can focus on details specific to their piece of hardware. Of the domain specific languages that have been implemented for this purpose, all are standalone tools which generate C code. None of these tools are widely used, and they suffer from several drawbacks. They require developers to learn a new syntax or GUI and they do not offer general programming-language features, so they can be difficult to extend. Also, because they must be executed, they introduce another dependency in the build process. This paper will describe an alternate approach, in which domain-specific code generation is implemented in a C++ library. C++ is a general purpose programming language which has some meta-programming facilities which allow it to mimic features of many DSLs. The first part of this paper describes the features offered by existing DSLs, and the second describes some C++ techniques that can be used to build a library for implementing device drivers.

II. Related Work

Consel et al [1] designed and implemented a language called Devil (DEVice Interface Language) which generates low level driver code for use in Linux and SunOS. With the Devil language, a programmer specifies how to access hardware registers for a device, and identifies bit ranges within those registers which can be treated as distinct variables. The Devil compiler then generates a series of C functions which read and write to the variables, and the driver code can call these functions to receive status information and send commands to hardware.

Since reads and writes to hardware registers have side-effects, Devil allows the programmer to specify conditions under which variables should be cached. And, to allow for cases when many variables need to be read of written at the same time, Devil allows variables to be grouped into structures, generating functions to read and write entire structures at once.

To deal with complicated hardware schemes for accessing registers, the driver writer may specify actions that will occur before and after register reads and writes. The more complicated access schemes can make use of private variables in actions, virtual registers, which map to other registers, and parameterized registers that take integral parameters for use in pre and post actions.

One of the major goals of Devil, aside from making driver code easier to read and write, is to provide safety. Each variable that the programmer declares can be constrained by a series of enumerated constants or by a bitmask. When a value is read or written which does not satisfy the constraints an error message can be printed at runtime. (Due to C's weak type system, Devil cannot always ensure that C code accessing variables is correct)

Manolitzas [2] describes a DSL for implementing Linux network drivers. His language is essentially to be a superset of Devil. But instead of being an interface language which generates C helper functions, it is an imperative language which can be used to generate an entire driver. The language requires the driver writer to specify at least 5 functions: init, open, transmit, receive, and stop, as well as an interrupt handler. The functions and the handler are written in a Pascal-like imperative language that has special features for reading and writing to hardware variables, managing memory, and performing synchronizations. Unfortunately, he does not describe the features for memory management and synchronization in any detail.

DSLs need not be text based. Compuware Driverworks provides a set of GUIs that can generate driver stub code for Windows platforms. The purpose of this product is to simplify the portions of driver code that interact with the operating system, rather than the parts that interact with the hardware.

The C++ library described in this paper attempts to provide developers with simpler ways of accessing the hardware in the same way that Devil does.

III. C++ Techniques

In some sense, solving a problem with C++ is the exact opposite of solving a problem with a DSL. DSLs are small, limited purpose languages while C++ is a huge, general purpose one. But while these differences may mean a lot to the people implementing domain specific solutions (since developing a C++ library is a different process than writing a compiler), the end goal is nearly the same. The ultimate goal is to allow the end user to write a small amount of expressive code to replace large amounts of repetitive or complicated code.

The use of C++ Libraries to implement domain specific solutions is nothing new. Blitz++ and POOMA, two pioneering C++ numeric libraries that perform domain specific optimizations on high level numeric code are now more than 5 years old. Other code-generating C++ libraries like parser generators and finite state machine emulators are being actively developed. By comparison this library, which generates instructions to access registers from variable accesses, is simple and even straightforward.

Past efforts at building specific DSL-like libraries in C++ have led to the discovery of generally useful techniques for C++ code generation. Czarnecki and Eisenecker [3] describe many of these techniques in detail, in the context of how they can be used to emulate features in real DSLs.

The rest of this section will provide an overview of how this library for writing device drivers will be used and of two C++

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

  // index variable
  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);
}
```

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

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

// registers for low and high bits
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;
```

Figure 1. The left column shows the Devil hardware description for a Logitech mouse driver. The right column shows the same description as a series of C++ type definitions. Example based on [1]

techniques that will be used in the library's implementation: typelists and the "curiously recursive template" pattern.

Registers and hardware variables which are described using language constructs in Devil will be described with type declarations in C++. Figure 2 shows some of these the C++ type declarations and the equivalent Devil statements which partially describe the hardware interface to a Logitech mouse. Each variable and register that is used to interact with the hardware corresponds to C++ type which is specified by the user by parameterizing generic `Variable` and `Register` template classes. The resulting Variable types can be instantiated as objects which can be read from or assigned to. There should be no reason to ever instantiate a Register type.

Many of the parameters for the `Variable` and `Register` Template classes are lists of arbitrary length, instead of individual values. These lists are called as typelists. The most comprehensive and easy to understand description of typelists was written by Alexandrescu in [4], but typelists are also mentioned in [1]. Typelists are made up of a chain of `Node` types where each `Node` type contains an arbitrary type and the type of a successor Node. The last node has a successor type of `Null`, where Null is just a special placeholder class. Figure 2 shows the specific definition for the `Null` type and the `Node` template class. Figure 3 shows how to declare a list of three types (int, signed int, unsigned int) using Node and Null classes. This notation is verbose and cumbersome, because it requires that template parameters be deeply nested. It is possible to provide a shorthand syntax using a template class that accepts a variable number of parameters. This shorthand is also demonstrated in Figure 3.

```
struct Null;

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

Figure 2. Node template class and Null types are used to make typelists.

Direct typelist declaration:

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

Shorthand typelist declaration

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

Figure 3. How to declare a list of three types, using the `Null` and `Node` classes directly, and by using `List` shorthand.

The real power of typelists comes from the fact that they can be manipulated and used to generate classes and values using compile-time algorithms. Figure 4 shows a simple algorithm class called `Length` that determines how many elements are in a typelist passed to it as a parameter. The length template class is specialized for the Null type to give a length of 0. It is specialized for any Node type to give a length of 1 plus the length of the Node's successor list. So when it is passed `MyList` it will give a length of (1 + (1 + (1+ 0))) = 3.

```
template<typename LIST>
struct Length;

template<>
struct Length<Null>
{
  enum { value = 0 };
};

template<class T, class U>
struct Length< Node<T, U> >
{
  enum { value = 1 + Length<U> };
};

cout << Length<MyList>::value;
```

Figure 4. Definition and use of the Length algorithm.

Length is one of the simplest typelist algorithms. Other commonly used algorithms return classes which inherit from every type on the list or returned sorted or filtered versions of lists. There are entire libraries filled with algorithms for manipulating typelists, including Boost::MPL (MetaProgramming Library) and Loki.

Lists are one type of parameter that can be passed to Variable and Register template classes in order to influence their behavior. Integer values are another type of parameter that can be passed in. Abstract base classes are a third type of parameter. In Figure 1, the abstract base parameters used are `Index` and `Mask`.

Abstract base classes are normally able to call methods on the classes which inherit from them. By default this works in C++ and some other object oriented languages through virtual function calls. Virtual function calls are problematic in this case, however, because C++ does not support virtual calls on templated functions. Additionally, virtual calls only allow abstract base classes to access their descendants' member functions, and not their internally defined constants and type definitions. A solution to both of these problems comes in the "curiously recursive template" pattern, also known as Barton and Nackman trick. This trick works by turning the abstract base class into a template class which takes the type of the descendant class as a single parameter. This way, when the base class needs to call a function on its descendant, it can cast its `this` pointer to the descendant type, and proceed to invoke the correct method. Figure 5 shows an example of this technique taken from Veldhuizen [5].

```
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> {
  ...
};
```

Figure 5. The curiously recursive template technique.

---

[1] 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.

[2] Apostolos Manzolitas. A Specific Domain Language for Network Cards. 2001. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/reports/apostolos.pdf

[3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patters Applied*. Addison-Wesley, 2001.
Techniques for Scientific C++

[5] Todd Veldhuizen. Techniques for Scientific C++. Indiana University Computer Science Technical Report #542, August 2000.
http://osl.iu.edu/~tveldhui/papers/techniques